NCC Group Whitepaper

Return of the Hidden Number Problem

June 13, 2018 -- Version 1.0

Prepared by
Keegan Ryan --- Senior Security Consultant

Abstract
Side channels have long been recognized as a threat to the security of cryptographic applications. Implementations can unintentionally leak secret information through many channels, such as microarchitectural state changes in processors, changes in power consumption, or electromagnetic radiation. As a result of these threats, many implementations have been hardened to defend against these attacks. Despite these mitigations, this work presents a novel side-channel attack against ECDSA and DSA. The attack targets a common implementation pattern that is found in many cryptographic libraries. In fact, about half of the libraries that were tested exhibited the vulnerable pattern. We implement a full proof of concept against OpenSSL and demonstrate that it is possible to extract a 256-bit ECDSA private key using a simple cache attack after observing only a few thousand signatures. As far as we are aware, the target of this attack is a previously unexplored part of (EC)DSA signature generation, which explains why mitigations are lacking and the issue is so widespread. Finally, we give estimates for the minimum number of signatures needed to perform the attack and suggest countermeasures to protect against this attack.
# Table of Contents

1. **Introduction** ................................................................. 3  
   1.1 Contributions ............................................................. 3  
   1.2 Attack Scenario .......................................................... 3  
   1.3 Related Work ............................................................... 4  

2. **Mathematical Background** .............................................. 6  
   2.1 ECDSA and DSA .......................................................... 6  
   2.2 The Hidden Number Problem ......................................... 7  
   2.3 Prior Attacks on ECDSA and DSA ................................. 8  

3. **Software Implementations** ............................................. 10  

4. **Cryptanalysis** .............................................................. 13  
   4.1 Chosen-Plaintext Attack ............................................... 14  
   4.2 Known-Plaintext Attack ............................................... 15  

5. **Experimental Results** ................................................... 17  
   5.1 Attacking OpenSSL ..................................................... 17  
   5.2 Solving the Hidden Number Problem .............................. 18  

6. **Countermeasures** ........................................................ 22  

7. **Conclusion** ................................................................. 23  

8. **Acknowledgments** ....................................................... 24
Introduction

Cryptographic systems are incredibly important for the security of modern technology. They are used to protect traffic on the internet[DR08][YL06] as well as banking information and government documents[NSS+17]. As cryptographic keys are used to protect higher value targets, it becomes even more critical that the keys are protected from compromise. Whenever a private key or a secret key is used, the owner risks having information about that key leak to their adversary, so cryptographic systems must defend against this threat.

There are countless ways that information can leak from a privileged context to an unprivileged one, and these are known as side channels. Side channels can arise from the timing differences caused by memory caches[OST06][Per05] or differences in branch prediction after running sensitive code[AKS07][EPAG16]. The recent Spectre[KGG+18] and Meltdown[LSG+18] vulnerabilities could be used to leak sensitive information past privilege boundaries and do so by abusing speculative execution and out-of-order execution respectively. Side-channel attacks are not just limited to microarchitectural state either: electronic devices can leak information via electrical signals[GPP+16a] or even the acoustic emanations of vibrating electrical components[GST14].

Cryptographic algorithms are often implemented with these attacks in mind, but the threat of a side channel inadvertently exposing cryptographic information is constant.

Contributions

In this work, we identify a new side-channel attack against ECDSA, DSA, and other ElGamal style signatures. This attack uses a previously known analysis method to recover the private key, but it targets a previously unexplored source of leakage: a single modular reduction in the signature calculation. We also explore two modes of the attack. The first is a chosen-plaintext attack, where the attacker is allowed to select the message being signed, and the second is a known-plaintext attack where the attacker has knowledge of the message being signed. We give success probabilities for different parameters of both modes of attack. We also show that this issue is widespread and that the attack may be used to extract keys from several common libraries. We demonstrate the feasibility of this attack by using a cache attack to recover a private key from a recent version of OpenSSL, and finally we give recommendations for how this attack can be prevented.

Attack Scenario

This work makes a few assumptions about the environment being attacked. In a vulnerable environment, the victim uses a private key to create several signatures. The attacker observes the resulting signatures and knows the messages being signed. Additionally, the attacker needs to be able to use a side channel to determine the outcome of a particular modular reduction in the signature process. This sort of setup reflects realistic attack scenarios.

Consider the case of a cloud hosted server using TLS and an ECDSA private key[DR08]. A passive attacker can observe a client's TLS handshake with the server, receiving an ECDSA signature in the ServerKeyExchange message. This signature includes randomness from the client and the server, as well as the server Diffie-Hellman parameters. All of these are transmitted in the clear, so the passive attacker has access to all the necessary values except the side-channel information, which may be obtained in multiple ways. If the attacker can also get assigned to a co-located virtual machine in the cloud environment[RTSS09][IGI+16], they can recover side-channel information about the victim by performing a Prime+Probe[OST06] attack. The attacker hopes to recover the server private key and then impersonate the server.

Similarly, consider the case of an attacker with a low-privileged account on an SSH server. Like TLS, the SSH Transport Layer Protocol[YL06] may use ECDSA to authenticate traffic, but unlike TLS, the signature is over a value based on a Diffie-Hellman shared secret, so a passive adversary would be unable to compute the message being signed and mount the attack. The low-privileged attacker, however, can actively initiate
a number of SSH connections, derive the shared secret, and compute the message that was signed by
the server. The attacker can simultaneously use the low-privileged account to mount a side-channel attack
against the SSH daemon process and leak the necessary information. The attacker then recovers the ECDSA
key and is free to impersonate the host and capture the credentials of a more privileged user.

Note that in both of these scenarios, the message being signed was not in the attacker’s control. During
signing, this message is transformed into an integer using a cryptographic hash, so this integer is uniformly
distributed over the possible range. This corresponds to the known-plaintext attack. However, in some
cases the attacker has more influence over the signed message. Consider a smart card or trusted execution
environment that signs a message of the user’s choosing; here, the attacker can select a message that, when
hashed, results in an integer that will leak the desired information through the side channel. Since only a few
uppermost bits of the integer need to have a certain value for this attack, the attacker can easily brute force
an input message that will hash to an appropriate value.

The signatures must also be randomized in this attack scenario. This is the case for non-deterministic ECDSA
and DSA, but it turns out that the attack is also likely to succeed in the deterministic case[Por13]. Non-
deterministic ECDSA and DSA require a randomly generated value, but in deterministic ECDSA and DSA
this value is derived from the signed message. As long as the signed messages are distinct, the derived
value should be indistinguishable from random choice. This means that in the chosen-plaintext attack, the
attacker may simply brute force a new input message so the hashed message still has the desired form,
but the derived value is completely different. The attacker could also reuse the same message so the same
value is derived, and then collect multiple measurements to reduce the effect of side-channel noise. In the
known-plaintext attack, it is likely the message varies anyway since there is not much utility in a system that
computes the exact same signature every single time.

These requirements apply to many systems. The signatures can be gathered passively or actively, and the
attacker may take advantage of a wide variety of side channels to leak the desired information. Both the
chosen-plaintext and known-plaintext attacks share a similar analysis, and only a few thousand signatures
are needed to recover the private key. Though it is impossible to enumerate all vulnerable scenarios, it is
clear that the risk extends beyond TLS and SSH servers, and many systems using ECDSA or DSA keys are
potentially susceptible to this key-extraction attack.

Related Work
This work is similar in nature to many recent publications. There is a long history of side channels being
used to compromise the integrity of a cryptographic library. RSA key generation in OpenSSL was recently
targeted with a Flush+Reload attack in [AGM+18], and a Prime+Probe attack was used in [DDME+18] to
target the EPID protocol in Intel’s SGX quoting enclave. Flush+Reload was also used in [GB16] to target
OpenSSL’s modular inversion routine for ECDSA.

These recent attacks target varied sources of leakage, but historically, attacks frequently targeted modular
exponentiation or scalar multiplication routines. Such issues affected both RSA[BBG+17] and Ed25519[GVY17]
in Libgcrypt, and the latter attack also demonstrated that non-constant modular reduction can be used as
a source of leakage. OpenSSL has been impacted by timing and cache side-channel attacks several times,
with vulnerabilities found in both modular exponentiation[YGH17] and scalar multiplication routines (See
[FWC16], [vdPSY15], [BvdPSY14], [YB14], [BT11], and [BH09]).

Other side channels have been explored as well. Power analysis was used in [GPP+16b] to target ECDSA
scalar multiplication on mobile devices, and against smart cards in [DMHMP13]. Branch prediction side
channels have been used to target OpenSSL’s implementation of modular exponentiation[AKS06][AKS07]
and modular inversion[AGS07].

As a whole, these previously published attacks demonstrate that side-channel attacks pose a real threat to cryptographic implementations; these attacks were frequently capable of recovering private keys through a variety of methods. This new work follows in the well established footsteps of this previous research.
Mathematical Background

Throughout this work we will use the following notation. Addition, multiplication, and division have their usual meaning when done over the reals. $a^{-1}$ is used to represent modular inversion in $\mathbb{F}_q$ where the modulus $q$ is large, prime, and clear from context. Logarithms are performed with base 2, and $[a, b]$ represents the range of integers no less than $a$ and no greater than $b$.

The reduction of a value modulo $q$ into the range $[0, q-1]$ is represented by $\lfloor \cdot \rfloor_q$. The absolute value modulo $q$, denoted $|\cdot|_q$, is found by reducing the argument into the range $[-\frac{q}{2}, \frac{q}{2}]$, then taking the absolute value.

Finally, we use $\text{MSB}_{l,q}(x)$ to denote knowledge about the $l$ most significant bits of $x$. That is, use $\text{MSB}_{l,q}(x)$ to denote an integer $u$ satisfying $|x - u| < q/2^{l+1}$.

This notation is both convenient and intuitive, since when $l = 0$, all values $u \in [0, q-1]$ satisfy this inequality, when $l = 1$, only about the $q/2$ values closest to $x$ satisfy the inequality, and so on.

ECDSA and DSA

ECDSA and DSA are two similar NIST standardized signature schemes\cite{KSD13}. Both are based on the ElGamal signature scheme\cite{Tah85} and consist of a large prime $q$ with bitlength $N = \lceil \log q \rceil$. The signature process also requires a function $F : [0, q-1] \rightarrow [0, q-1]$ that is difficult to invert. In (EC)DSA, this function uses exponentiation over a finite abelian group and relies on the difficulty of the (Elliptic Curve) Discrete Logarithm problem to make the signature secure. Unlike previous work, the details of $F$ are not necessary to understand our attack.

The (EC)DSA signature scheme also uses a hash function $\text{Hash} : \{0, 1\}^* \rightarrow \{0, 1\}^{\text{outlen}}$ to convert an input message to an output of fixed bitlength, which can be encoded as an integer. The choice of hash function depends on bitlength $N$, but is selected so that the hash output has enough bits of security to match the desired level of security of the signature\cite{Bar16}.

When signing a message, the signer, in possession of private key $x \in [1, q-1]$, performs the following steps:

1. Calculate $m = \text{leftmost min}(N, \text{outlen})$ bits of $\text{Hash}(msg)$
2. Generate nonce $k$ uniformly from $[1, q-1]$.
3. Calculate $r = F(k)$. If $r = 0$, return to step 2.
4. Calculate $s = \lfloor k^{-1} (m + r \times x) \rfloor_q$. If $s = 0$, return to step 2.
5. Return signature $(r, s)$.

For the purposes of this paper, the details of the corresponding verification algorithm are also unimportant.

When $\text{outlen} \geq N$ and $q \approx 2^N$, then $m$ is roughly uniformly distributed in $[0, q-1]$. This is the case for many common choices of $q$ and $\text{Hash}$, including the use of SHA-256 with NIST Curve P-256 and SHA-384 with Curve P-384 found in NSA Suite B Cryptography\cite{SH12} and the recommended curve and hash associations found in RFC 5480\cite{THP+09}. However, there are still some valid combinations where $\text{outlen} < N$. In these cases, $m$ is not uniformly distributed in $[0, q-1]$. Take for example the standardized curves over binary fields, whose orders have bitlengths that do not match the output size of common hash functions. Consider Curve B-283, which has an order of length 282 bits and therefore a security strength of 128 bits. SHA-256 is a suitable hash function at this security level, and so messages signed with this valid combination will have

$$m < 2^{256} < 2^{282} = q$$
and so the uppermost 26 bits of \( m \) will always be zero.

These combinations of curves and hash functions are not frequently used, and so the remainder of this work makes the simplifying assumption that \( m \) is uniformly distributed in \([0, q - 1]\). This makes the attack easier to analyze while still being applicable to many important cryptographic libraries. Although not as common as the use of Curve P-256 or Curve P-384, the use of Curve P-521 with SHA-512 is notable since it also breaks the uniformity assumption, but it turns out there is an easy modification that can make the attack succeed in this case. This is explored in Section 4.

The Hidden Number Problem

Though not originally developed for the purpose of cryptanalysis [BV96], an important tool for understanding cryptanalytic attacks on ECDSA and DSA is known as the Hidden Number Problem (HNP). There are many similar constructions of the HNP, but here we present the more general formulation and solution algorithm as found in [BvdPSY14].

The HNP poses the problem of recovering an unknown scalar in a prime field when only partial information is known about multiples of the scalar. Formally in the HNP, one has knowledge of a number of multipliers \( t_1, \ldots, t_u \in \mathbb{F}_q^* \) for known prime \( q \). For each sample \( i \), one also knows \( \text{MSB}_{i,q}(\lfloor t_i \alpha \rfloor_q) \) for some fixed but unknown \( \alpha \in [1, q - 1] \). The goal is to recover hidden number \( \alpha \). The HNP can identically be expressed as a system of inequalities:

\[
|\lfloor t_i \alpha \rfloor_q - u_i|_q < q/2^{l_i+1} \quad \text{for all} \quad i \in \{1, \ldots, d\}.
\]

Boneh and Venkatesan recognized in [BV96] that the HNP has much in common with the study of lattices. If we let \( b_1 = (t_1, \ldots, t_d, 1) \) be a basis vector of a lattice, then vector \( \alpha b_1 = (\alpha t_1, \ldots, \alpha t_d, \alpha) \) will be relatively close to vector \((u_1, \ldots, u_d, 0)\) after some multiple of \( q \) is subtracted from each component of \( \alpha b_1 \). The concept of subtracting a multiple of \( q \) is encoded for each dimension by another basis vector \( b_{i+1} = (\ldots, 0, q, 0, \ldots) \).

Thus given an instance of the HNP, we can construct a lattice basis so that one of the vectors is close to \((u_1, \ldots, u_d, 0)\) and at a distance in each dimension of no more than \( q/2^{l_i+1} \). Recovering hidden number \( \alpha \) is therefore possible once this vector is found. In fact, this so-called Closest Vector Problem (CVP) can be solved efficiently using the LLL lattice basis reduction algorithm [LLL82] and Babai’s nearest plane algorithm [Bab86]. [BV96] additionally showed that when the \( t_i \) are uniformly and independently drawn, the \( l_i \) are sufficiently large, and enough HNP inequalities including \( \alpha \) are known, a probabilistic algorithm exists to solve the HNP in polynomial time.

This choice of lattice basis vectors is not the only option, however, and there have been several different representations based on the same idea as [BV96] (See, for example [NS02][NS03]). One drawback of the original construction is that all \( l_i \) are identical, since the CVP solver weighs all dimensions equally, and cannot incorporate information that the bound on some HNP inequalities might be tighter than others. The representation in [BvdPSY14] has the added advantage of not requiring that all \( l_i \) are the same, and it does so by scaling the dimensions of the lattice proportionally to the inverse of the bounds.

In their construction, which we use here, we construct lattice \( L(B) \) from the basis vectors given by the rows of matrix \( B \):

\[
B = \begin{bmatrix}
2^{l_1+1}q & 0 & \cdots & 2^{l_d+1}q & 0 \\
\vdots & \ddots & \ddots & \vdots & \vdots \\
2^{l_1+1} & \cdots & 2^{l_d+1} & 0 & 1
\end{bmatrix}.
\]

We also set vector \( u = (2^{l_1+1}u_1, \ldots, 2^{l_d+1}u_d, 0) \). As before, note that there exists some vector \( x \in \mathbb{Z}^{d+1} \) such
that \( xB = (2^{t_1+1} | t_1 \alpha |_q, \ldots, 2^{t_d+1} | t_d \alpha |_q, \alpha) \), and this lattice vector is close to vector \( u \) as guaranteed by the HNP inequalities. We hope that solving the CVP with lattice \( L(B) \) and vector \( u \) will yield lattice vector \( xB \), since the last coordinate is the hidden value \( \alpha \).

Babai’s nearest plane algorithm is not the only way to solve the CVP. In fact, it is possible to use what is called the embedding strategy[NS00][Won15] to reduce an instance of the CVP to one of the Shortest Vector Problem (SVP). The general embedding strategy for lattice \( L(B) \) and vector \( u \) is to construct a new lattice \( L(B') \) from the rows of matrix

\[
B' = \begin{bmatrix} B & 0 \\ u & q \end{bmatrix}
\]

and running a lattice basis reduction algorithm on \( L(B') \). The second smallest vector in the reduced basis is often of the form \( (xB - u, -q) \), and once again the last coordinate of \( xB - u \) is \( \alpha \).

Other lattice basis reduction algorithms besides LLL may be used. Although it does not provide the same polynomial time guarantees as LLL, BKZ[SE91] with a block size of 15 to 20 has performed well on instances of the HNP[BvdPSY14]. An improved version of BKZ, called BKZ 2.0[CN11], implements even more features such as advanced pruning.

The lattice-based method to solving the HNP is practical, efficient, and easy to implement. It requires relatively few HNP inequalities to correctly recover the hidden number, runs in polynomial time under certain parameters, and is likely to give the correct answer. However, this method struggles when the number of bits \( l_i \) per inequality is not large enough[NS02], and the method is not very resilient to errors. Fortunately, the use of lattices is not the only practical way to solve the HNP. Bleichenbacher’s solution to the HNP, as documented in[DMHMP13], uses exponential sums to detect the influence of small biases. Bleichenbacher’s approach is more tolerant of collection errors and can recover the hidden number when only a small amount of information is leaked per inequality, but this comes at the cost of requiring many more samples than the lattice method. As long as it is possible to pose a cryptanalytic problem as an instance of the HNP, the tools exist to find the solution.

**Prior Attacks on ECDSA and DSA**

The use of the HNP as a tool in a cryptanalytic attack against ECDSA and DSA implementations dates back to the work of Howgrave-Graham and Smart[HGS01]. Their attack assumes that, in addition to knowing the signature values, an attacker knows a proportion of the bits in nonce \( k_i \) for each of several signatures. This knowledge is encoded into an instance of the HNP, and solving yields the secret private key \( x \).

To see how this works, assume the attacker collects \( d \) signatures \((r_i, s_i)\) and knows \( \text{MSB}_{l_i, q}(k_i) \) for \( i \in \{1, \ldots, d\} \). There are many ways the attacker can gain this information about \( k_i \). For example, the information may leak through a side channel[FWC16][vdPSY15][BvdPSY14][YB14][BT11][BH09] or be induced via an injected fault[NNTW05] that clears known bits. Using the equations for ECDSA and DSA signatures, the attacker’s knowledge can be reformulated to the following (dropping index \( i \) for simplicity).
\[ u = \text{MSB}_{l,q}(k) \]
\[
\frac{q}{2^{i+1}} > |k - u|
\]
\[
= \left| \left\lfloor s^{-1} (m + rx) \right\rfloor_q - u \right|_q
\]
\[
= \left| \left\lfloor \left\lfloor s^{-1}r \right\rfloor_q x \right\rfloor_q - \left\lfloor u - s^{-1}m \right\rfloor_q \right|_q
\]

This gives a single HNP inequality, and collection of many signature samples yields more inequalities, giving an instance of the HNP. Solving the HNP yields hidden number \( x \), from which new signatures may be forged.

Other formulations exist where the attacker has knowledge of a block of contiguous bits in the middle of \( k \) or a combination of most significant and least significant bits[\text{HGS01}]. This method can even be applied when non-contiguous groups of bits are known[\text{HR07}], but these require more HNP inequalities to be collected for the problem to be solved.
Though there are many ways to implement the algorithm in Section 2.1 and compute ECDSA and DSA signatures, a common one is presented in Algorithm 1. Many helper functions are needed to fully implement these operations, but for brevity, only the Mod and Sign functions are shown here. This is sufficient to demonstrate the relevant parts of the implementation and to identify two implementation patterns that, if present, indicate the implementation is susceptible to the key extraction attack.

Algorithm 1 Create (EC)DSA Signature

1: function Mod(a, q) ▷ Return the value of a reduced modulo q
2:   if a < q then
3:     return a
4:   else
5:     quotient, remainder ← DivRem(a, q) ▷ Get remainder after dividing by q
6:     return remainder

7: function Sign(msg, x, q) ▷ Compute a signature over msg using private key x
8:   m ← Hash(msg)
9:   m ← Mod(m, q)
10:  k ← RandomInteger(1, q − 1)
11:  ki ← Inv(k, q)
12:  r ← F(k)
13:  if r = 0 then
14:     return error ▷ Unlikely to ever be reached.
15:  rx ← Mul(r, x)
16:  rx ← Mod(rx, q)
17:  sum ← Add(m, rx)
18:  sum ← Mod(sum, q)
19:  s ← Mul(ki, sum)
20:  s ← Mod(s, q)
21:  if s = 0 then
22:     return error ▷ Unlikely to ever be reached.
23: return (r, s)

Two things are important to note here. First, Mod does not run in constant time. If the argument being reduced is already in the range \([0, q − 1]\), this function returns early without calling DivRem. If a side channel reveals whether DivRem was called by this function, then the attacker learns information about argument \(a\). Even though many of the analyzed (EC)DSA implementations do not follow this pattern exactly, most include some logic so the Mod function returns early if the argument is already reduced. A minority of implementations are written so the Mod function executes in constant time, regardless of input.

Secondly, observe the modular reduction on line 16. This ensures that the product \(rx\) has been reduced into the range \([0, q − 1]\) prior to addition with \(m\). In some implementations, this reduction before addition, or less frequently the reduction following addition, is omitted.

Several common open source implementations of the ECDSA, DSA, and ElGamal signing algorithms were analyzed to observe which implementation pattern was followed. If the Mod function leaks range information about the argument and the product \(rx\) is reduced, the implementation is probably vulnerable to the attack. Due to the sheer number of potentially affected libraries, implementations were analyzed via static review of source code. However, to demonstrate that the attack is possible under such conditions, a full
proof of concept was developed for one of the implementations, described in Section 5.1. The results of the analysis are shown in Table 1.
<table>
<thead>
<tr>
<th>Library</th>
<th>ECDSA</th>
<th>DSA</th>
</tr>
</thead>
<tbody>
<tr>
<td>CryptLib[Pet]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>LibreSSL[liba]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mozilla NSS[Moz]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Botan[Llo]</td>
<td>Yes</td>
<td>NR</td>
</tr>
<tr>
<td>OpenSSL[opea]</td>
<td>Yes</td>
<td>NR</td>
</tr>
<tr>
<td>WolfCrypt[Wol]</td>
<td>Yes</td>
<td>NR</td>
</tr>
<tr>
<td>Libgcrypt[Gnu]</td>
<td>Yes</td>
<td>NR</td>
</tr>
<tr>
<td>LibTomCrypt[libc]</td>
<td>Yes</td>
<td>NR</td>
</tr>
<tr>
<td>OpenJDK Libsunec[Opeb]</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>MatrixSSL[mat]</td>
<td>Yes</td>
<td>N/A</td>
</tr>
<tr>
<td>BoringSSL[Goo]</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>BouncyCastle[bou]</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>Crypto++[cry]</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>Golang crypto/tls[gol]</td>
<td>NR</td>
<td>NR</td>
</tr>
<tr>
<td>C#/Mono[csrn]</td>
<td>NR</td>
<td>N/A</td>
</tr>
<tr>
<td>mbed TLS[mbc]</td>
<td>NR</td>
<td>N/A</td>
</tr>
<tr>
<td>Trezor Crypto[Tre]</td>
<td>NR</td>
<td>N/A</td>
</tr>
<tr>
<td>Nettle[M]</td>
<td>No</td>
<td>NR</td>
</tr>
<tr>
<td>BearSSL[Por]</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>Libsecp256k1[libb]</td>
<td>No</td>
<td>N/A</td>
</tr>
<tr>
<td>NaCl[BLS]</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**Table 1:** Cryptographic implementations of (EC)DSA signatures. The table examines several popular cryptographic libraries to see if they follow the vulnerable pattern. If modular reduction both precedes and follows the addition of $m$ and the modular reduction routine is not constant time, the entry is marked as a “Yes.” If the product $rx$ or the sum $m + rx$ is not reduced, then the entry is marked as “NR” because it is not vulnerable to this attack, but information may still be leaking during arithmetic operations involving $x$. Finally, “No” means the arithmetic operations are constant time, and “N/A” means that algorithm is not implemented.
As is hinted at in the previous section, the new key extraction attack targets information leaked by non-constant time modular reduction in the computation of signature component $s$. Specifically, the attack targets the modular reduction on line 18 in Algorithm 1. This reduction follows the addition of $m$ to $\lfloor r \cdot x \rfloor_u$. Of course, both terms of the sum are reduced into the range $[0, q - 1]$, so therefore $m + \lfloor r \cdot x \rfloor_u \in [0, 2(q - 1)]$. Within this range, the side-channel leakage in MOD thus reveals if $m + \lfloor r \cdot x \rfloor_u \in [0, q - 1]$.

The possible values can be constrained further, since SIGN does not return a signature unless $r$ and $s$ are nonzero. Since $r \neq 0$, $\lfloor r \cdot x \rfloor_u \in [1, q - 1]$, and since $s \neq 0$, $m + \lfloor r \cdot x \rfloor_u \notin \{0, q\}$.

Assume the side channel perfectly reveals the truth value of $m + \lfloor r \cdot x \rfloor_u \in [0, q - 1]$. If this is true, then the attacker concludes

$$m + \lfloor r \cdot x \rfloor_u \in [0, q - 1] \Rightarrow \lfloor r \cdot x \rfloor_u \in [−m, q − m − 1] \cap [1, q − 1] = [1, q − m − 1]$$

$$\Rightarrow |\lfloor (q − r) \cdot x \rfloor_u − \frac{m}{2}|_q < \frac{q − m}{2}$$

and gains an HNP inequality for hidden number $x$. Similarly, if the side channel reveals that $m + \lfloor r \cdot x \rfloor_u \notin [0, q − 1]$, then the attacker concludes

$$m + \lfloor r \cdot x \rfloor_u \notin [0, q − 1] \Rightarrow m + \lfloor r \cdot x \rfloor_u \in [q + 1, 2(q − 1)]$$

$$\Rightarrow \lfloor r \cdot x \rfloor_u \in [q − m + 1, 2q − 2 − m] \cap [0, q − 1] = [q − m + 1, q − 1]$$

$$\Rightarrow |\lfloor (q − r) \cdot x \rfloor_u − \frac{m}{2}|_q < \frac{m}{2}.$$
In practice, the side channel may not be perfectly accurate. The attacker processes the side-channel data and guesses the truth value of $m + [rx]_q \in [0, q - 1]$, but this processing may suffer from false positives or false negatives. The lattice approach used to solve the HNP is not resilient to errors, so an incorrect guess at this stage prevents the analysis from being successful.

To surmount this difficulty, the attacker must design the side-channel processing to have either a low false positive rate or a low false negative rate. Often the attacker can choose between one or the other but cannot have both. Thus if the false positive rate is low, then the attacker can confidently include the derived HNP inequality when the processing indicates $m + [rx]_q \in [0, q - 1]$ (and similarly for the false negative rate). If the false positive rate is low and the processing indicates $m + [rx]_q \notin [0, q - 1]$, the attacker does not know for sure if the statement is true since the false negative rate may be high. When the processing indicates this, the attacker discards the sample.

The error rate only needs to be low enough that the set of derived inequalities is unlikely to contain any errors. More advanced error handling methods can be found in [BT11], but this sort of analysis exceeds the scope of this work. We shall instead make the simplifying assumption that either the false positive or false negative rate is zero.

For the remainder of this section, we shall assume the false negative rate is zero; that is, all inequalities are of the form $\left|\left((q-r)x\right)_q - \frac{m}{2}\right| < \frac{m}{2}$. We also let $\mu$ represent the probability that $m + [rx]_q \notin [0, q - 1]$ is correctly detected as such. This makes it slightly easier to describe the chosen-plaintext and known-plaintext attacks, and the derivation is not substantially different when the false positive rate is zero.

**Chosen-Plaintext Attack**

In the chosen-plaintext variation, the attacker is able to choose the most significant bits of $m$. There is a trade-off present for choosing a large or small $m$. The larger $m$ is, the more likely it is that $m + [rx]_q \notin [0, q - 1]$ and the more likely it is that the attacker can construct an inequality based on a given collected signature. However, a smaller $m$ is a tighter bound on the inequality, and so each inequality reveals more information about $x$. Additionally, the lattice-based method to solving the HNP is less effective on looser bounds. The attacker must take all of these into consideration when selecting the desired $m$.

We may thus provide an estimate of how the number of signatures that need to be observed scales with the attacker’s choice of $m$. We make several simplifying assumptions and approximations, including the assumption that the information from one HNP inequality is independent from that of the others. With this in mind, each constraint for the attacker’s $m$ provides about $\log \frac{q}{m}$ bits of information about $x$. Since $x$ has about $\log q$ bits in total, roughly $\frac{\log q}{\log q - \log m}$ successfully collected samples are needed. The odds of a given signature yielding a usable constraint is $\frac{\mu m}{q}$, giving the expected number of signatures to collect as about

$$\frac{q \log q}{\mu m (\log q - \log m)}.$$

This simple analysis suggests that the fewest signatures are needed when $m = q/e$. However, note that the lattice-based HNP solver may have trouble since each sample contains less than 1.5 bits of information about $x$, so it is likely that a better value will be slightly higher and can be found through experimentation.

The chosen-plaintext attack also has an interesting application in the case of Curve P-521 used with the SHA-512 hash function. Here, $q$ is 521 bits, but $m$ is at most 512 bits. Thus if the side channel reveals $m + [rx]_q \in [0, q - 1]$ with no false negatives, then the chosen-plaintext attack can be used with $m =
$2^{512} \approx q/2^9$. This gives an expected number of required signatures of

$$\frac{q \log q}{\mu m (\log q - \log m)} \approx \frac{2^9 \times 521}{\mu (521 - 512)} \approx 30000/\mu.$$  

This is still on the order of tens of thousands of signatures, so in many cases this attack on P-521 is practical.

**Known-Plaintext Attack**

In the known-plaintext variation, the attacker is only able to observe $m$, not influence it. The attack proceeds much the same way as the chosen-plaintext attack, but the attacker must decide which values of $m$ to collect samples for and which to ignore. Similar to [BT11] and [BvdPSY14], the attacker selects a threshold value $m_t$. The attacker then collects signatures until the attacker collects a sample where $m < m_t$ and $m + \lfloor rx \rfloor \notin [0, q - 1]$. The attacker adds the derived inequality and repeats the process until $d$ inequalities have been recovered. Finally, the attacker uses the lattice-based method to solve the HNP and recover private key $x$.

There is once again a trade-off at play, since a smaller value of $m_t$ yields more information about $x$ in each sample, but a larger value of $m_t$ increases the odds that a sample is collected.

We can once again estimate how many samples need to be collected as a function of $m_t$. The probability that $m < m_t$, $m + \lfloor rx \rfloor \notin [0, q - 1]$, and the side channel reveals this information is approximately

$$P(m < m_t \text{ and } m + \lfloor rx \rfloor \notin [0, q - 1] \text{ and detected})$$

$$= \mu P(m < m_t \text{ and } m + \lfloor rx \rfloor \notin [0, q - 1])$$

$$= \mu \sum_{m'=0}^{m_t-1} \frac{1}{q} P(m + \lfloor rx \rfloor \notin [0, q - 1] | m = m')$$

$$\approx \mu \sum_{m'=0}^{m_t-1} \frac{m'^2}{q^2} \approx \frac{\mu m_t^2}{2q^2}.$$  

Since the conditional distribution of $m$ is uniform in $[0, m_t - 1]$, the expected number of bits of information about $x$ per sample is about

$$E_m[- \log \frac{m + 1}{q}] = \sum_{m=0}^{m_t-1} \frac{-1}{m_t} \log \frac{m + 1}{q}$$

$$= - \frac{q}{m_t} \sum_{m=1}^{m_t} \frac{1}{q} \log \frac{m}{q}$$

$$\approx - \frac{q}{m_t} \int_{0}^{m_t/q} \log x dx$$

$$= - \frac{q}{m_t} \frac{m_t}{q} (\log \frac{m_t}{q} - \log e)$$

$$= \log \frac{eq}{m_t}.$$  

Combining this with the odds of collecting a usable sample and the fact that about $\log q$ bits of $x$ must be
recovered, the expected number of signatures to collect is approximately

\[ \frac{2q^2 \log q}{\mu m_t^2 \log(eq/m_t)}. \]

This is minimized when \( m_t = q/\sqrt{e} \), or when about 0.7 bits are leaked by each HNP inequality. Once again, the lattice-based HNP solver may have issues with such values, so the actual optimal value of \( m_t \) is likely determined by what the solver is able to solve.

We may also perform the known-plaintext attack in the case of Curve P-521 and SHA-512. Similar to the chosen-plaintext attack, we can set \( m_t = 2^{512} \approx q/2^9 \). Since \( m < m_t \) for all \( m \), the expected number of samples does not quite follow the above derivation, and is given by

\[ \frac{2q \log q}{\mu m_t \log(eq/m_t)} \approx \frac{2^{10} \times 521}{\mu(\log e + 9)} \approx 50000/\mu. \]

Once again, the attack is still practical for Curve P-521, despite the mismatch between the hash size and curve order.
The attack consists of both a collection phase and an analysis phase, so two experiments were performed to demonstrate the feasibility of the full attack. In the first experiment, the collection phase was studied by examining OpenSSL 1.1.0g[opea] and using a Flush+Reload attack[YF14] to determine the outcome of the modular reduction. This is used to give probabilities describing the oracle behavior of an example implementation and to show that a co-resident attacker can reliably use this side channel to perform an attack.

The second experiment explored the analysis phase by determining how the analysis performs under different parameters. Rough estimates are given for the minimum number of signatures needed for multiple common key sizes.

**Attacking OpenSSL**

In order to demonstrate the feasibility of the attack, a Flush+Reload attack was performed on the most recent version of OpenSSL. The attack targets version 1.1.0g, which is shipped with Ubuntu 18.04, and the attack was performed on an Intel i7-6600u CPU.

Flush+Reload is a side-channel attack where the attacker and the victim share the same memory. The attacker repeatedly flushes particular addresses from the cache and monitors the time to reload the address. If the memory at the address remains evicted, the access time is high, but if the victim accesses the memory between the flush and the reload, the memory access time is low. Due to the copy-on-write behavior of the Linux kernel, an unprivileged attacker process maps the same shared library as a privileged victim process, so the attacker is capable of detecting when the victim loads particular code at the 64 byte L1 instruction cache line granularity. The Flush+Reload attack is implemented by the Mastik library[Yar16], which was used for this attack.

The experiment targeted the libcrypto.so.1.1 shared library, which contains the ECDSA signing code of the OpenSSL library. Five offsets within the library were monitored:

1. 0x0f4e70 - Entry point of ECDSA_do_sign_ex
2. 0x0f4582 - Call to BN_mod_add_quick when computing \( m + \lfloor rx \rfloor_q \)
3. 0x0a4b9f - End of BN_usub, called from BN_mod_add_quick when the sum of arguments exceed the modulus.
4. 0x0a4b42 - Middle of BN_usub
5. 0xed180 - End of ECDSA_SIG_free.

The first offset was monitored in order to detect when the signature process began so the attacker could collect Flush+Reload samples until the last offset was reached. The second offset was used to detect when \( BN\_mod\_add\_quick \) was called, which happened close to the end of the trace, and to observe if the third or fourth offsets were triggered at that point as well. Two offsets within \( BN\_usub \) were monitored because this had the effect of reducing noise and giving the cleanest signal. By running this collection process many times, it was possible to analyze how the side channel behaves when \( m + \lfloor rx \rfloor_q \in [0, q - 1] \). The collection was run for 100000 samples using a known private key and randomly generated \( m \), and the following results were observed:

\[ \text{SHA-256 a13c42ae2e12dc0cb9aba3133ff0db2e8dfada69d2ca5f4e399d4be08c1e14677} \]
The remaining 5449 samples did not follow the expected pattern of the first, second, or fifth offset, and are excluded from the table. From these results it is clear that this side channel may be used as an low false positive oracle: when neither the third nor fourth offsets were hit, \( m + \lfloor rx \rfloor \) \( \in [0, q - 1] \) with observed frequency \( 1 - 4/33927 \approx 99.99\% \). We can also estimate \( \mu = \text{Prob}(3^{rd} \text{ and } 4^{th} \text{ offsets not hit}|m + \lfloor rx \rfloor \in [0, q - 1]) \approx 0.68 \).

Using this side channel, the full attack can be tested. A privileged process was set up that listened to a localhost port, signed incoming messages using \texttt{libcrypto.so} and a 256-bit ECDSA private key, and returned the signatures. The attacker was implemented in an unprivileged process and was set up to perform the chosen-plaintext attack with \( m = q/2^l \). The attacker triggered signatures while performing the Flush+Reload attack and repeated until 100 usable samples were collected. Then \texttt{FPLLL[dt16]}, a library which implements lattice operations, was used to perform the BKZ reduction with a block size of 20. Based on \( \mu \), it should take 4706 signatures to collect enough samples for the HNP. In fact, over a sample size of 30 runs, it took on average 4465 signatures. For this test, the parameters were chosen arbitrarily so that the lattice reduction succeeds. The attack generalizes to other parameters as well, including larger key sizes or different values of \( l \). The next section explores which parameters to choose for best performance.

### Solving the Hidden Number Problem

In addition to demonstrating the practicality of the side channel, it is important to give parameters for which the attack performs well. While it is easy to determine the expected number of signatures one needs to gather before having enough HNP inequalities, it is less easy to determine how many inequalities are needed. Estimates for this number are derived in Section 4, but this is done after making several simplifying assumptions and does not capture the actual process of solving the HNP. This data is collected by simulating the side-channel attack and running the lattice-based HNP solver several times.

Both the chosen-plaintext and known-plaintext attacks were tested for orders of size 256 and 384, the sizes recommended by NSA Suite B Cryptography. 160 bit and 521 bit orders were also considered in order to capture the behavior at both allowed extremes. For each test case, a random prime order of the selected size was generated, and a parameter \( l \) was chosen, roughly corresponding to the number of bits leaked per signature. In the chosen-plaintext attack \( l \) was used to set \( m = q/2^l \) and in the known-plaintext attack it set \( m_t = q/2^l \). To determine how many HNP inequalities to include, which sets the dimension of the lattice, an overhead factor \( f \) was used. Thus for the chosen-plaintext attack, \( \lceil f \log q \rceil \) HNP inequalities are generated, and for the known-plaintext attack, this value is \( \lceil f \log q \rceil + \log q \lceil f \log q \rceil \). Based on the estimate, the attack is expected to fail for \( f < 1 \) and expected to succeed for \( f > 1 \). This formed the instance of the HNP.

A reduction method was chosen at random to attempt to solve the instance of the HNP. Specifically, after using the embedding strategy to encode the HNP as a basis for a lattice, the \texttt{FPLLL} library used its implementation of either LLL or BKZ to reduce the lattice. Block sizes of 10, 15, 20, and 25 were used for BKZ. After reduction completed or a timeout of five minutes was reached, the best guess was retrieved and

\footnote{In the case of the 521 bit modulus, \texttt{FPLLL} failed to ever terminate when using BKZ. This could be due to a bug in the reduction library, so the BKZ results for that modulus size are omitted.}

<table>
<thead>
<tr>
<th>3\textsuperscript{rd} offset hit?</th>
<th>4\textsuperscript{th} offset hit?</th>
<th>Sum in ([0, q - 1])</th>
<th>Sum not in ([0, q - 1])</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>y</td>
<td>243</td>
<td>46861</td>
</tr>
<tr>
<td>y</td>
<td>n</td>
<td>1261</td>
<td>16</td>
</tr>
<tr>
<td>n</td>
<td>y</td>
<td>11811</td>
<td>428</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>33927</td>
<td>4</td>
</tr>
</tbody>
</table>
Figure 1: Performance of LLL reduction algorithm when solving the HNP for a 521 bit modulus. Reduction fails for $f < 1$ and succeeds for $f > 1.3$, which aligns with our expectations of the estimate of the number of inequalities needed.

compared to the actual $x$ used to generate the data. The time taken and outcome were also both recorded. Experiments were performed on an Amazon EC2 C5.18xlarge instance.

In [BvdPSY14], time is used when considering the performance of a set of parameters, but in this work, minimizing the number of signatures is prioritized. In real life systems, the attacker may be limited by the number of times they can observe the signature method but have substantial time to perform the actual reduction. Since this experiment limited reduction time to five minutes, unless the attacker needs to solve the HNP faster than this, the results will still apply.

The experiments reveal that the estimates derived in Section 4 are accurate in most cases. Figure 1 demonstrates this, as it is clear reduction fails for $f < 1$ and succeeds for $f > 1.3$. There is a transition region between 1 and 1.3 where the probability of success is in between 0 and 1. Assuming the understanding of the information leaked per sample is correct, an ideal reduction algorithm would quickly transition between low probability of success and high, since as soon as there is enough information, the ideal algorithm would be able to solve the problem. However, LLL and BKZ are not ideal algorithms, and the observed transition is more gradual.

This slope between success and failure becomes more flattened as $l$ decreases, as can be seen in Figure 2. For $l = 3$, the curve appears completely flat, indicating that there is a minimum number of bits that must leak for reduction to be successful. Since the expected number of signatures is exponential in $l$, the optimal value will be when $l$ is as low as possible, but not so low that reduction is unlikely for reasonable values of $f$.

Next, consider how the attack performs for different reduction algorithms. In Figure 3 it can be seen that as the BKZ block size increases, so does the success ratio. These results indicate that increasing block size improves the reduction algorithm’s ability to recover the hidden number from a limited number of samples; however, increasing block size comes at the cost of time. This was especially the case for tests of the 384 bit modulus, since the larger matrices timed out past the five minute cutoff when using a block size of 25.

Putting all this together provides the following approximate minimum bounds for a 90% success rate, given
Figure 2: Performance of BKZ reduction algorithm when solving the HNP for a 256 bit modulus. This data is for the chosen-plaintext attack, and the BKZ block size is 15. For each value of $l$, the reduction fails for $f < 1$. However, for $f > 1$, the odds of success rise at different rates.

Figure 3: Performance of various reduction algorithm when solving the HNP for a 256 bit modulus. This data is for the chosen-plaintext attack with $l = 4$. As the block size increases, the minimum overhead ratio needed for good performance decreases.
<table>
<thead>
<tr>
<th>Key size (bits)</th>
<th>Chosen-Plaintext Attack</th>
<th>Known-Plaintext Attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>160</td>
<td>$l = 3$, BKZ-25</td>
<td>530</td>
</tr>
<tr>
<td>256</td>
<td>$l = 4$, BKZ-25</td>
<td>1300</td>
</tr>
<tr>
<td>384</td>
<td>$l = 5$, BKZ-25</td>
<td>3000</td>
</tr>
<tr>
<td>521</td>
<td>$l = 9$, LLL</td>
<td>40000</td>
</tr>
</tbody>
</table>

Table 2: Approximate parameters for the minimum signatures required. These values performed the best of the combinations that were tested, and they gave above a 90% success rate. Under these parameters, the reduction step reliably finished in under five minutes. Here, BKZ-25 is shorthand for BKZ reduction algorithm with a block size of 25.

In general, the chosen-plaintext attack requires fewer signatures total, but the known-plaintext attack can use a smaller value of $l$. Overall, the number of signatures needed is fairly low, making this both a practical and a powerful attack.
We propose two countermeasures to mitigate this particular attack. The first approach is to use constant time arithmetic operations when computing $s$ from $m$, $r$, $x$, and $k$. This will eliminate timing-based side channels from leaking sensitive state at this stage. Although only replacing the modular addition with a constant time alternative suffices to prevent the particular attack identified in this work, a similar attack may apply to non-constant time modular multiplication. If a function leaked the approximate size of the reduced product or factors, an attacker could mount a similar HNP-style attack using knowledge of small $[rx]_q$ or $[m + rx]_q$. As the signature computation is dominated by the point multiplication to compute $r$, the final stage of computing $s$ is a minority of the total time taken. The cost of replacing these operations with their constant time alternatives is unlikely to incur a significant penalty on signature time. However, this approach only attempts to mitigate timing and control-flow based side channels, which may not be sufficient to protect against all attacks, such as power analysis. For many libraries which were built around general purpose multiple precision integer libraries, migrating to a constant time implementation may also be difficult.

Another approach is to use blinding to reduce the usefulness of side-channel information. A similar method of blinding can be used to harden RSA signature generation, elliptic curve point multiplication, and modular exponentiation. For this approach, the signer selects a random $b \in [1, q - 1]$. The signer then computes $[bx]_q$ and $[bm]_q$. After adding these two values together, the signer multiplies by $b^{-1}$, giving a value of $b^{-1}(bm + bx)_q = [m + rx]_q$. The remainder of signature computation proceeds as normal. Assuming $b$ is kept secret and the attacker has access to the same side channel, the attacker will learn the truth value of $[bm]_q + [bx]_q \in [0, q - 1]$. However, this exposes no information about $x$.

To see this, first consider the case $m = 0$. $[bm]_q + [bx]_q = [brx]_q \in [0, q - 1]$, regardless of the secret key, so no information is leaked. Next, we will show that when $m \neq 0$, changing the sign of $b$ changes the truth value of $[bm]_q + [bx]_q \in [0, q - 1]$. Thus since $[1, q - 1]$ is the union of disjoint $(b, [-b])$ pairs, exactly half of the values in $[1, q - 1]$ make the statement true, and the other half make it false, regardless of $x$, $m$, or $r$. Thus the side channel reveals no information about the private key. Take $b \in [1, q - 1]$ such that $[bm]_q + [bx]_q \in [0, q - 1]$ is true. Therefore,

$$[bm]_q + [bx]_q \in [0, q - 1]$$

$$\iff [bm]_q + [bx]_q < q$$

$$\iff q - [-bm]_q + q - [-brx]_q < q$$

$$\iff - [-bm]_q - [-brx]_q < -q$$

$$\iff [-bm]_q + [-brx]_q > q$$

$$\iff [-bm]_q + [-brx]_q \notin [0, q - 1]$$

with the last step relying on the fact that since $s \neq 0$, $[-bm]_q + [-brx]_q \neq q$.

Although it is tempting to conclude from this proof that blinding is a sufficient mitigation to protect against this class of attack, there are some caveats. The proof assumes that the only information that leaks is whether or not the sum exceeds $q$, but the non-constant time implementation of modular reduction may leak extra information that invalidates the mitigation. It is also possible that information about $b$ may leak through another side channel, such as during the modular inversion of $b$, again reducing the effectiveness of the countermeasure. While blinding stops the basic attack, it is no guarantee that non-constant time code cannot be exploited in other ways. More sophisticated exploitation, however, is not the focus of this work.
Conclusion

This work only explores a small aspect of this issue, and there are many interesting areas of exploration that remain. One such question is whether this vulnerability can be exploited through timing data alone. The non-constant implementation of modular reduction does introduce a slight difference $\delta$ in timing between samples that have $m + \lceil rx \rceil \not\in [0, q - 1]$ and those that do not, but recall that this attack requires either a low false positive rate or a low false negative rate to succeed. If we let $\tau$ be the random variable representing the time it takes to create a signature where $m + \lceil rx \rceil \not\in [0, q - 1]$, then we can model the total time taken as $T = \tau + \delta 1_{m + \lceil rx \rceil \not\in [0, q - 1]}$. Here, $1_X$ represents the indicator variable that equals 1 if $X$ is true, and 0 otherwise. If $\tau$ has high variability, then it will be difficult to correctly guess the outcome of the reduction based on $T$ alone.

One potential solution to this problem is to find situations where $\tau$ has sufficiently low variance. One of the major sources of variation in a modern computer is cache behavior, because even though the same instructions are being executed, their duration depends on processor state, which varies from run to run. However, it is possible that embedded systems without caches may have sufficiently low variation that a pure timing attack is possible on these platforms. Somewhat ironically, for security reasons, common (EC)DSA implementations attempt to make scalar multiplication constant time with respect to secret data, which reduces variance even further.

Another potential solution is to use a more error-tolerant approach to solving the HNP, such as Bleichenbacher’s solution[DMHMP13]. In this approach, the problem is posed as a set of pairs $(c_i, h_i)$ for which the set $\{\lceil h_i + c_i x \rceil \not\in [0, q - 1] \}^d$ is biased towards the values 0 and $q$. The remainder of the attack recovers $x$ from this bias. Consider (with a slight abuse of notation) the distribution of

$$\left\{ \left\lfloor -\frac{q}{4} - \frac{q}{2} m_i + \lceil r_i x \rceil \not\in [0, q - 1] + r_i x \right\rfloor \right\}^d_{i=1}.$$

As it turns out, this distribution is also biased towards 0 and $q$. Intuitively, this is because a large $\lceil r_i x \rceil$ is more likely to have $m_i + \lceil r_i x \rceil \not\in [0, q - 1]$. Thus we expect that the pairs $\left( -\frac{q}{2q} T_i, r_i \right)$ will similarly exhibit a consistent bias, although this bias will be incredibly small due to the bias of $\tau$. It is an open question whether Bleichenbacher’s solution can practically be applied here, but it suggests the possibility that a passive adversary monitoring a large number of signatures could recover the private key from timing data alone.

We have only considered a small number of contexts in which this issue might be exploited. There are countless other side channels besides Flush+Reload that could leak the desired information, including branch predictor side channels[ERAG+18][AKS07] or power analysis[KJJ99]. This work identifies a common but flawed implementation pattern that, when paired with a sometimes simple side channel, can be exploited to disclose the private key in an attack that is both simple and practical. There are likely several other unidentified systems that are similarly affected by this issue and for which key compromise is possible. Finally, this work reinforces the importance of constant time implementations and hardening against side channels, as it emphasizes and demonstrates the dangers that even a small leak can pose to a cryptographic system.

23 | Return of the Hidden Number Problem NCC Group
I would like to thank Thomas Pornin for his insightful technical review of early drafts, and I would also like to thank David Wong, Andy Grant, and Audrey Saunders for their helpful feedback on this paper. Their thoughtful reviews have helped shape the work into its current state, and it would not be the same without them.
References


[bou] BouncyCastle.


25 | Return of the Hidden Number Problem  NCC Group

[124x665] [cry] Crypto++. 12  
[124x644] [csm] C# mono. 12  
[dt16] The FPLLL development team. fplll, a lattice reduction library. Available at https://github.com/fplll/fplll, 2016. 18  
[gol] Go crypto/tls. 12  
[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. Rsa key extraction via low-bandwidth acoustic


[KSD13] Cameron F. Kerry, Acting Secretary, and Charles Romine Director. FIPS PUB 186-4 Federal Information Processing Standards publication Digital Signature Standard (DSS), 2013. 6

[liba] LibreSSL. 12
[libb] Libsecp256k1. 12
[libc] LibTomCrypt. 12


[mat] MatrixSSL. 12

[mb] mbed TLS. 12


[NNTW05] David Naccache, Phong Q. Nguyên, Michael Tunstall, and Claire Whelan. Experimenting with


[opea] OpenSSL.

[Opeb] OpenJDK. Libsunec.


[Por] Thomas Pornin. BearSSL.


