Extracting Private Keys from Enclaves through a Controlled-channel attack on (EC)DSA Signing

Kaushik Kasi
ksk@berkeley.edu
ID: 26340592
Year: 4th-Year Undergrad
Project type: Research/Implementation

December 1, 2018

1 Introduction

Enclave computing has gained popularity in recent years due to its promise of allowing secure operations to execute in untrusted computing environments. Intel SGX has been at the forefront of the trusted computing advancement, and there has already been a lot of effort to port secure computation from a regular computing context, like cryptographic libraries, into the enclave context. At a high level, enclaves provide trusted computing by introducing logical and memory barriers between untrusted host code and trusted enclave code, which is signed.

Since enclave programs run within an unprotected hosts environment, any data leakage has the potential to be instrumented, captured, and analyzed at length. This becomes especially problematic in cryptographic contexts where data leaks can reveal information about a fixed and secure private key. In this project I mounted a controlled-channel attack on ECDSA signing, which in many SSL implementations, conditionally uses an overflow function that leaks a bound about the private signing key [6]. Since the binary of the enclave library is known, the attack takes advantage of this fact by modifying the host OS and SGX driver to revoke read access on the memory page where overflow function exists, note if that fault happened during the signing process, and keep track of associated inputs and outputs.

This function can be called many hundreds or thousands of times within the host program to further bound the key. By using a lattice reduction algorithm on a system of constructed inequalities related to when the fault occured, we can solve for the private key. As far as I have tested it, the attack is able to always (with 100% success rate) solve for 16-bit private keys given enough samples. This marks an improvement over previous work, which have used cache-timing based attacks that are not always reliable.

2 Background

2.1 Intel SGX

As mentioned earlier trusted computing platforms like Intel SGX have been showing a lot of promise in areas like cryptography and privacy-sensitive applications. In this attack, we are most concerned with how enclave programs are loaded in memory and subsequently run, as enclave resources share hardware with the host operating system. The goal with this subsection

is to give a background about SGX in order to understand the feasibility of a memory-based controlled-channel attack.

At a high level an enclave application consists of two complementary parts. One part is the enclave program itself, which is somewhat restrictive with what data types and libraries it can use. It runs in a protected and encrypted region of memory called the Enclave Page Cache [1] and protects the confidentiality and integrity of its code and data from the host operating system. The host operating system can call into the enclave using a predefined API ECALL. SGX also introduces 18 new processor instructions, implemented in microcode, that handle operations like creating an enclave, paging in/out encrypted memory, and exiting the enclave during an interrupt. Memory for an Enclave program is stored in an Enclave Page Cache (EPC), which is an area of shared DRAM memory that is encrypted by the Intel Memory Encryption Engine (MEE)[1].

It is important to note that all memory accesses are first sorted out by a page table that is common to both the trusted and untrusted environments, and also controlled by the untrusted operating system. To this extent, the untrusted OS can control how pages of the enclave program are mapped, and even the read/write permissions of the memory pages that the enclave could access, This opens up the possibility for a controlled-channel attack, assuming the host has access to the enclave program before it is loaded.

Once an Enclave program is loaded into memory, the host OS can only interface with it using a fixed ECALL/OCALL interface that the enclave has defined in a .EDL file. For example, if the enclave program had a function that signed messages, the message would be passed into the program with an ECALL from the host C or C++ program, and the return value from the enclave is written to a buffer that the host has preallocated. The attacked enclave application in this project uses exactly this interface.

2.2 SSL and (EC)DSA Signing

SSL Libraries like OpenSSL provide a lot of cryptographic functionality including key generation, signing, and verifying. This project specifically concerns (EC)DSA signing which is a common way of veriffying the integrity and identity of messages. The algorithm is described below, and is based on a figure in [6].

```
function SIGN(msg, x, q):
    m = hash(msg)
    m = mod(m, q)
    k = rand_int(1, q-1)
    ki = mod_inverse(k, q)
    r = F(k)
    rx = mul(r, x)
    rx = mod(rx, q)
    sum = add(m, rx)
    sum = mod(sum, q)
    s = mul(ki, sum)
    s = mod(s, q)
    return (r, s)
```

It is important to note that the terms m, r, and q are actually public values that are known to the attacker, and x is the private key that lives in the enclave and is unknown to us, which this attack is trying to infer. M is a certain number of upper bits of the Hash of the message passed into the program, R is a nonce value, a hash of a random value that is generated in the signing process and returned to us. Q is a large prime number modulus, which is public

as it is required for verifying the signature [10]. As we will show in following sections, the use of a certain overflow function in the SSL signing process within the mod function indicates a bound relating m, r, q, and x. We can use enough of these bounds to eventually solve for x. Furthermore, Intels current version of SGX-SSL [4], which implements ECDSA signing to the enclave context, is vulnerable to a controlled-channel attack that would leak whether the overflow function was called.

2.3 Mathematics of Private Key Leakage during (EC)DSA Signing

As noted by the NCC paper, many SSL implementations conditionally use an overflow function in their modulus operation when the public modulus q is smaller than the argument. In other words, the modulus function often looks like the below figure (code based on a snippet in [6]),

```
function MOD(a, q):
    if a < q:
        return a
    else:
        return DIV_REM(a, q)
```

where DIVREM is called if $a >= q$. If we can observe that DIVREM is called, we can draw a conclusion about the hidden private key x. This math was first worked out in [6].

$$\begin{gathered} m + [rx]_q \notin q \\ m + [rx]_q \in [q+1, 2(q-1)] \text{ since } m < q \\ [rx]_q \in [q-m+1, q-1] \\ [(q-r)x]_q \in [1, m-1] \\ [(q-r)x]_q - \tfrac{m}{2} \in [1 - \tfrac{m}{2}, \tfrac{m}{2} - 1] \\ [[(q-r)x]_q - \tfrac{m}{2}]_q < \tfrac{m}{2} \end{gathered}$$

Note that while q is fixed, m is the hash of the message passed into the user, and r is a random value that is expected to change in each version of the signing algorithm. This means that every invocation of this signing function can provide a slightly different bound on the value of x when the overflow function is called.

The bounds that are discovered about x can be formulated under the hidden number problem, first described in [9], which can be solved using a lattice basis reducer. The hidden number problem involves a system of inequalities of the below form :

$$[[t\alpha]_q - u_i]_q < \tfrac{q}{2^{l+1}}$$

Bounding a hidden number α. Notice the similarity between the above expression and the bound on our private key

$$[[(q-r)x]_q - \tfrac{m}{2}]_q < \tfrac{m}{2}$$

Where:

$$\begin{aligned} t &= (q-r) \\ u_i &= \tfrac{m}{2} \\ \alpha &= x \text{ (our private key)} \end{aligned}$$

We can also force m to always equal $\frac{q}{2}$ by finding a message that when hashed, has the top k bits as $\frac{q}{2}$, if the implementation hashes the message within the signing operation.
In the NCC formulation of the problem, they find that one way to solve the above system is to use information from the above bounds to form a basis for a lattice, and then reduce that basis to recover the original alpha, or secret. The basis is constructed in this matrix below:

$$B' = \begin{pmatrix} B & 0 \\ u & q \end{pmatrix}$$

Where B embeds q and r collected from each run as such:

$$B = \begin{bmatrix} 2q & & & 0 \\ & \ddots & & 0 \\ & & 2q & 0 \\ 2(q-r) & 2(q-r) & \dots & 1 \end{bmatrix}$$

u embeds $-\frac{m}{2}$ which is actually just:

$$u = \begin{pmatrix} q & q & q & \dots & q \end{pmatrix}$$

Once this basis is reduced using the LenstraLenstraLovsz (LLL) Algorithm [2], the final row of the original matrix:

$$\begin{pmatrix} u & q \end{pmatrix}$$

where we embedded information about the message, once reduced will now be of the form

$$A = \begin{pmatrix} 2[(q-r)x] - \frac{q}{4} & 2[(q-r)x] - \frac{q}{4} & \dots & x & -q \end{pmatrix}$$

Where the private key x, second to the last value of this row, is solved for

2.4 Types of attacks on Intel SGX

In the period that Intel SGX has been around, attacks on Intel SGX have fallen into 3 main categories: branch prediction attacks, controlled channel attacks, and cache attacks [12]. The exact type of attack mounted will depend on how precisely data needs to be collected, though in this discussion, we will only look at controlled channel attacks.

2.4.1 Controlled Channel Attacks

This project uses a controlled-channel attack on the OS page table to page-fault on the page where the mod overflow function lives, which removes any uncertainty about whether DIVREM (the mod overflow function) was called. This kind of attack was described earlier in [12] and used in several other attacks on cryptographic procedures [8, 12] to recover private information from enclaves. Since the host OS controls the memory resource management, the host can map code pages into arbitrary locations of memory and fault on those pages to determine the control flow of the program.

This kind of memory leakage becomes even more effective when there are input-dependent control flows within the enclave program, because in many cases, combining known information about the parameters of an enclave program and being able to observe the control flow can partially or completely reveal information about secret data. For example, in [12] a page-faulting attack is implemented on EdDSA against both OpenSSL and Libgcrypt in an enclave

setting to recover around 27% of bits (on average) of a private key, by observing the subroutine access patterns related to scalar multiplication. More recently, the AsiaCCS paper [8] cleverly attacked RSA key generation to retrieve a complete private key by page faulting on subroutines within the recursive Euclidean GCD algorithm that indicated whether the key at iteration i was odd or even. These papers both provide solutions to obfuscate what a host OS might perceive, but it is very difficult to completely mitigate these attacks.

One may think that while page-faulting attacks are useful, they are severely limiting because they can only perceive control flows at a page-level granularity, but not be able to perceive individual instructions within a page being executed. Recently a paper [5] introduced a framework for conducting single-stepping attacks on an enclave to instrument an enclaves execution line-by-line. The framework accomplishes this by configuring an Advanced Programmable Interrupt Counter (APIC) to trigger and exit the enclave after every instruction, which requires saving the last executed instructions address into a preallocated State Save Area (SSA) within the enclave. This SSA is actually readable using a command called EDBGRD by the host, so the framework reads and saves this information after every instruction in order to reconstruct the exact execution that the program took. While this adds a significant overhead to the programs execution time, many side-channel attacks would be enabled by this kind of granularity. The side-channel exploited in this attack does not require single-stepping as the DIVREM function is compiled to live on a distinct page from the rest of the signing code. However this attack does use SGX-Step to instrument page faults and actually implement the attack, as I will describe later.

2.5 Comparisons and Contributions

There are two papers, Single Trace Attack Against RSA Key Generation in Intel SGX SSL [8], and Return of the Hidden Number Problem [6] which my work heavily compares and contrasts from. The method for key extraction, a page-faulting side-channel attack, was inspired by the Single Trace paper, although that project attacks the key generation stage. The mechanics and mathematics of attacking ECDSA key-signing in a non-enclave context were first written about in Return of the Hidden Number Problem.

2.6 Single Trace Attack Against RSA Key Generation in Intel SGX SSL

The AsiaCCS paper Single Trace Attack Against RSA Key Generation in Intel SGX SSL [8] attacks RSA key generation in the verification step where (p-1) and (q-1), corresponding to primes p, q in N = pq, are checked to see if they are coprime to each other using the recursive Binary Euclidean algorithm (BEA). At a high level, BEA takes in 2 arguments p and q, works by reducing the arguments by a factor of 2 and swapping the arguments p and q recursively. As the authors of the paper identified, the reduction of the arguments by 2 actually calls a different sub-function within OpenSSL depending on whether the argument in question is even or odd. The paper also uses SGX-Step, which is used in this project, to instrument which sub functions are called at each iteration and therefore deduce the internal state of each recursive function call. With the information about which arguments are even and odd in each function call, and the number of calls before termination, the attack is able to collect enough data in one invocation to solve a system for the 99% of the private keys bits [8]. The Single Trace Attack Against RSA Key Generation is more robust than mine because it only requires one run to recover most of the secret bits, and it exploits an input-dependent control flow that directly depends on the secret data, so there are fewer parameters to account for when solving. However it requires more reasoning at each iteration to reconstruct the state of the function at each time step.

2.7 Return of the Hidden Number Problem

Another related attack is described in the paper Return of the Hidden Number Problem [6]. As mentioned earlier, that paper discovers the information leakage linked to observing the mod overflow function, formalizes the bound that it places on the private key, and introduces a method to solve for that private key. However the implementation of the attack itself suffers from a few shortcomings. The attack uses Flush + Reload [13] to determine whether the overflow function had been called or not. In this attack a malicious process flushes the cache at a high interval and times how long it takes to load the overflow function to determine whether the SSL procedure had used it recently.

This attack isnt possible in the enclave context where trusted and untrusted code do not share the same memory cache. Furthermore, since Flush + Reload depends on cache timing data in an active system, there is an inherent amount of uncertainty about whether the DIVREM mod overflow was actually called. Since the attack works by placing bounds on the key and solving a system, even one spurious result can make the solving system inconsistent or find the wrong key. My attack uses a page-faulting method to determine if the DIVREM function was called, so it is possible to mount this attack in an enclave context. Since a page-fault is deterministic, meaning it wont trigger without the page actually trying to be accessed, there is no possibility of a false negative or a false positive being introduced.

To this end, my work introduces 2 new contributions:

1. My work is the first example of a proof of concept extracting an (EC)DSA key from within an enclave context by attacking the signing function
2. My attack does not suffer from false negatives or false positives as earlier works do. Therefore if there is sufficient data collected to recover the bits of the private key, my attack can solve for it with 100% accuracy and reliability

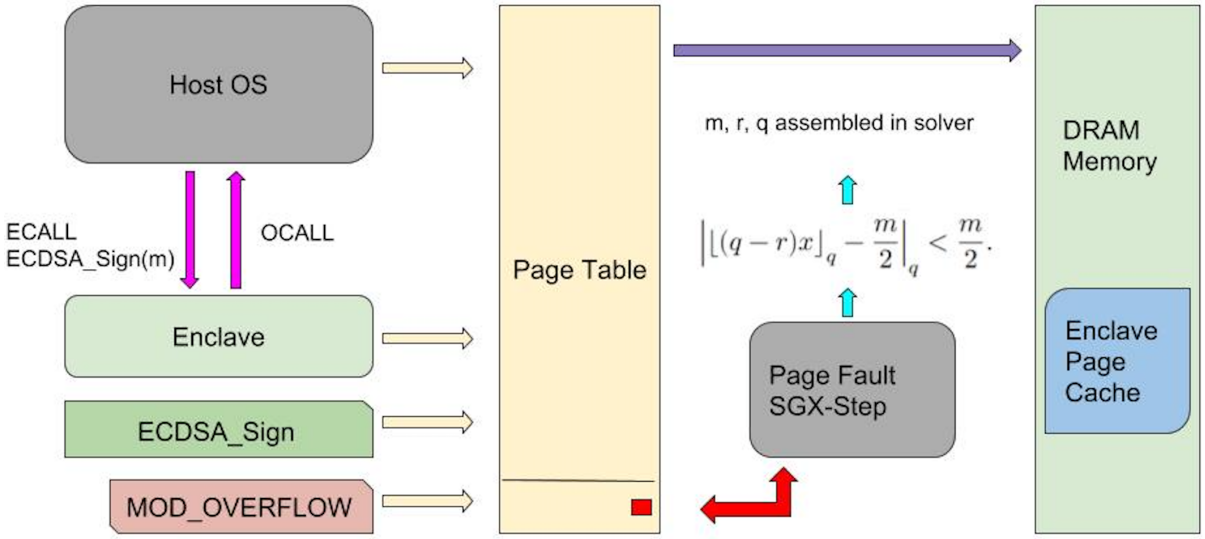
3 Problem Definition and Motivation

This attacks aims to extract information about an (EC)DSA private key that is stored within a trusted enclave from an untrusted context, by taking advantage of side-channel information. This problem is interesting to look at because this attack has never been carried out in an enclave context before, so it requires a new approach to extract the key. The latest version of SGX-SSL, which is the Intel implementation of OpenSSL for enclave computing, is vulnerable to this attack, so any results from this endeavor will have direct ramifications on the security of performing cryptographic operations within an enclave context.

4 Method and Approach of Attack

In this attack model I am assuming that we have full control over the host OS including its memory management. I am also assuming that we have access to the binary of the enclave (EC)DSA signing program, and that we can call it an unlimited number of times. This attack was performed on a Ubuntu laptop with an Intel Core i76500U. At a high level, there are 5 components that make this attack possible: the (EC)DSA signing enclave application, the calling application, the operating system that manages the memory and page table, a program that runs the program repeatedly to collect and saves the data, and finally a program that assembles the data into a lattice basis matrix and passes it into FPLLL to solve for the private

key. The whole system is shown below, and I will go into detail about each part.



4.1 (EC)DSA Enclave Signing Program

As shown earlier, the ECDSA signing procedure takes a message, hashes it, and truncates it into m. It performs some further arithmetic, and returns r, which is a hash of a random number generated in the procedure, and s which is the signature of the message. The private key is hard-coded into the enclave, which is what we are trying to recover from outside of the enclave. For the purposes of this attack I wrote an abridged version of (EC)DSA signing (partially shown below) which doesnt include s and any associated computation, as s is not required for this attack. I am also taking in the hashed value of the message instead of computing the hash value within the function. Whether the hash is computed inside the function or not is implementation specific, and it makes it a bit clearer to show the point, as otherwise we would have to search for an arbitrary message whose top bits would hash to $\frac{q}{2}$, which is required for our chosen plaintext attack.

```
1  unsigned long private_key_x;
2  unsigned long public_mod_q;
3
4  function SIGN(hashed_msg):
5      m = mod(hashed_msg, public_mod_q)
6      k = rand_int(1, public_mod_q-1)
7      r = F(k)
8      rx = mul(r, private_key_x)
9      rx = mod(rx, private_key_x)
10     sum = add(m, rx)
11     sum = mod(sum, public_mod_q)
12     return r
13
14 function MOD(arg, q):
15     if arg < public_mod_q:
16         return arg
17     else:
18         return DIV_REM(arg, public_mod_q)
```

Also note the implementation of mod and the conditional calling of DIVREM. We will eventually want to fault on the add operation (line 10) to move up to the part of the code

with the side-channel, then unblock ADD and fault on DIVREM afterwards to see whether the overflow function was called for $M + RX_q$.

MOD and DIVREM need to be on separate memory pages in order to discern when DIVREM is conditionally called, since this attack uses page faulting instead of single-stepping. For this to happen I compiled the enclave binary using *gcc* with $-O2$ and $-falign-functions = 4096$ flags, so that the functions are guaranteed to be on different memory pages (of size 4096 bytes). This will enable a page-faulting attack to discern whether DIVREM was actually called.

4.2 Calling Application

Customarily the enclave application ships with a calling application that runs in an untrusted environment, that calls functions inside of an enclave using a predefined secure interface defined in an .EDL file (shown below). In our case the EDL file has an interface for the calling application to pass in a (hashed) message and receive the associated r value from the signed application. It also includes helper functions for the calling application to ascertain the address of the Add, Mod, and overflow (DIVR) functions for mounting the page-fault attack, though in a real-world setting the host OS could get this information without the helper functions, since it has the binary. We instruct the calling function to always pass in a value of q/2 as the hashed message to the ECDSA_sign function, so that the received inequality is in the form that the solver can solve, as described earlier. However the returned r value is random and likely to be different in each iteration, and because the faulting behavior is dependent on m + rx, we can observe faulting on the DIVREM mod overflow function on some runs and not on others, even when m is kept the same. The calling application is also outfitted with utilities from SGX-Step to trigger and handle page faults.

```
enclave {
        trusted {
                        public long int ECDSA_sign(unsigned long int hashed_msg);
                        public void* get_Add_ADDR(void);
                        public void* get_Mod_ADDR(void);
                        public void* get_DIVR_ADDR(void);
    };
        untrusted {
                void uprint([in, string] const char *str);
        };
};
```

4.3 Host Environment with SGX-Step

This attack depends on page-faulting to uncover the information leak about the private key, and this requires memory pages to be marked as not readable/writeable prior to execution in order for a fault to trigger. Furthermore when the fault does trigger, we have to keep track of which functions page faulted and also unblock the page for execution to continue. This is not trivial to implement in kernel code, and so I took advantage of the SGX-Step [5] framework. SGX-Step modifies the kernel and the SGX driver to allow a user to define a custom fault handler function that gets called when a page-fault is triggered in the host OS. A pointer to the fault-handling function is pushed to the unprotected call stack automatically during the Asynchronous Enclave Exit (AEP) procedure when a fault is called, which gives a user the chance to note which address was faulted on and unblock the memory page so that execution can resume. SGX-Step also handles calling the ERESUME function after the fault was triggered and handled, so that the machine goes back to running the enclave again.

```
unsigned long public_mod_q;
bool add_caught = false;
bool mod_caught = false;

// Calls the enclave signing application
int main( int argc, char **argv )
{
        SGX_ASSERT( sgx_create_enclave( "./Enclave/encl.so") );

    register_aep_cb(aep_cb_func);
    register_enclave_info();

    get_Add_ADDR(eid, &add_ptr);
    get_Mod_ADDR(eid, &mod_ptr);
    get_DIVR_ADDR(eid, &ptr);

    // fault on add function
    // will fault on mod overflow once add is faulted
    ASSERT(!mprotect(add_ptr, 4096, PROT_NONE));

    unsigned long int message_by2 = Q / 2;
    unsigned long int return_v;
    ECDSA_sign(eid, &return_v, message_by2);

    // if execution triggered a fault
    if (mod_caught) {
      printf("DATA%lu:%lu:%lu\n", message_by2, return_v, Q);
    }
            SGX_ASSERT( sgx_destroy_enclave( eid ) );
        return 0;
}

void fault_handler(int signal)
{
    if (!add_caught) {
        add_caught = true;
        // Unblock the page with the add operation
        ASSERT(!mprotect(add_ptr, 4096, PROT_READ | PROT_WRITE));
        // Fault on the page with the mod overflow
        ASSERT(!mprotect(mod_ptr, 4096, PROT_NONE));
    } else { // if add was already caught, this fault corresponds to the mod overflow fn
        ASSERT(!mprotect(mod_ptr, 4096, PROT_READ | PROT_WRITE));
            mod_caught = true;
    }
}
```

As you can see from the above code, the main function of the calling application uses Linuxs mprotect to block read/write permission initially on the add functions page, so that as the attacker we know when that point of the logic is reached. Once the first fault is triggered, we note that add has been reached, unblock add using mprotect, and block the page pertaining to the DIVREM mod overflow function. If we fault again, we know that $m + rx_q > q$ and output the corresponding m, r, and q values since the bound holds. Note that in reality mod is actually called again later in the signing process, so in practice we have to use some additional information in the fault handler to ascertain whether the $m + rx_q > q$ overflow actually happened.

4.4 Data Collection

For this attack we repeatedly call this enclave with $m = q/2$ and observe the returned r and associated faulting behavior in order to infer bounds about the private key. I wrote a small bash script to call the enclave an arbitrary number of times, and then store the associated m, r, and public modulus q in a text file if faulting behavior is observed. The exact number of data points to recover a key of size n is around log q / (log q - log m) as found empirically by [6], and I have explored this experimentally further in the results section.

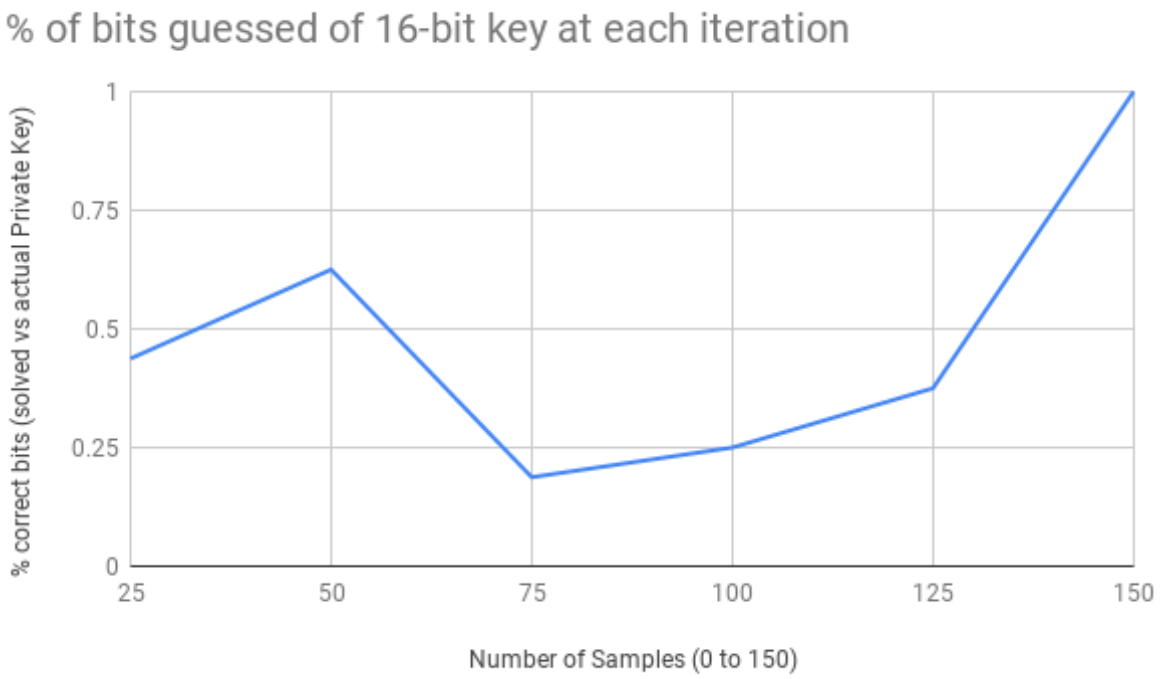
4.5 Data Analysis

The data collected from the enclave is loaded into a Python script on a separate, more powerful machine. The m, r, and q values are loaded into lists and from that, the B matrix (from the background section) is constructed using numpy. The u vector and larger lattice basis B matrix is constructed using numpys hstack and vstack functions. I am using the FPLLL library which implements the Lenstra Lenstra Lovasz algorithm to reduce a lattice basis matrix, which given enough samples, will solve the system for the final private key value.

5 Results and Evaluation

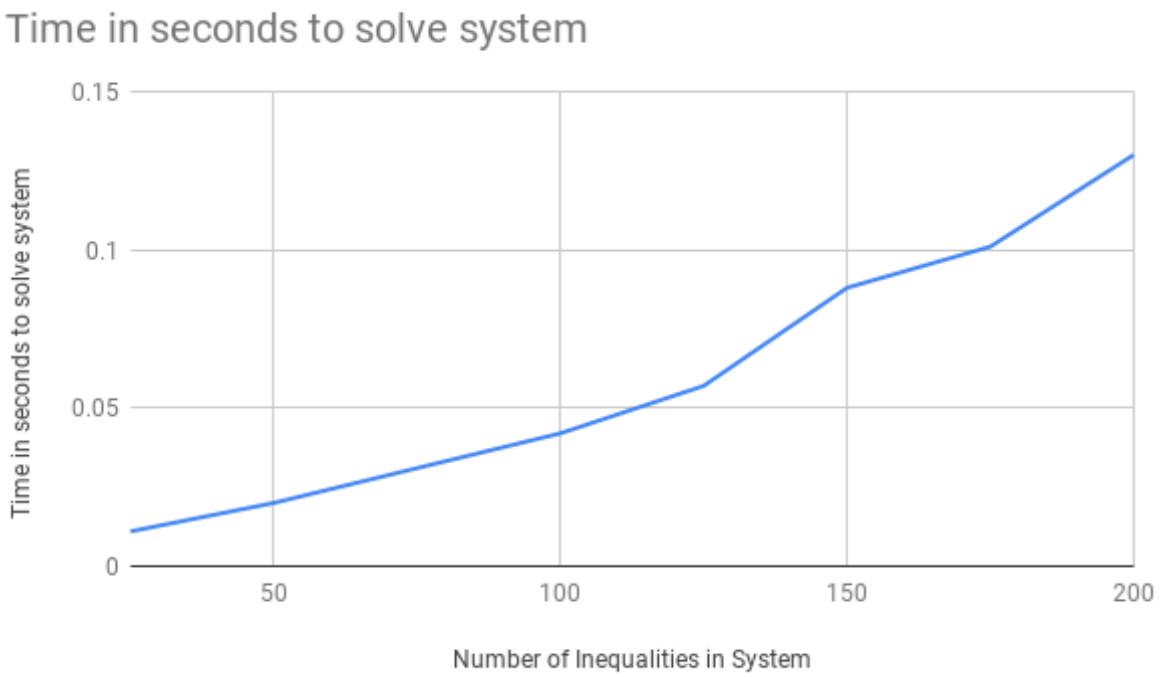
The experiment was run 5 times for a key of 16 bits, and in each experiment, the private key was recoverable. For each experiment I hardcoded a different private key and public modulus q into the enclave, ran the enclave repeatedly, and evaluated the enclave based on whether the private key could be solved for by the solver from the data collected. On average a key of 16 bits required 150 samples to completely recover.
The size of the private key will determine the number of samples it takes to recover the private key completely, and it will also determine the amount of time that it takes to solve for the system. I ran some preliminary experiments and found that 32-bit keys were not recoverable on my machine given 5,000+ samples and 5-10 minute execution times, though keys of up to 256 bits are solved for in [6] using a large AWS compute instance.
I recognize that most private keys in practice are 128 bits or larger, but the same mathematics and dynamics should apply.

5.1 Accuracy of bits



The above graph shows the bit-for-bit accuracy for a single run of the solver trying to recover a 16-bit key with the number of samples used increasing in increments of 25. The accuracy starts high, dips, and then reaches 100% correctness by the end of the evaluation. This is because from what I have observed, the solver tends to leave as many 0 bits as possible in the initial stages, and then makes finer (but possibly incorrect) guesses as it sees more samples. Further analysis into the LLL algorithm can possibly explain this phenomenon. For larger problems, it is very possible that a full solution is not feasible. It would be very helpful to isolate parts of the solved key that we are confident in so that the rest of the key can be brute-forced.

5.2 Time to Solve for Private Key



The time required to solve for the private key is roughly a function of the number of equations in the system, which is a function of the private key size. From what we can observe here we see a slight polynomial trend in the solving time as the number of samples increases. From preliminary experiments, solving for 32-bit keys took between 5 and 10 minutes but were not able to complete with a successful solution for the private key.

6 Future Work

This project opens up a number of new areas for exploration. Several earlier papers have shown successful side-channel attacks where a significant percentage or majority of bits were recoverable, and the rest were brute forced. Even when the LLL algorithm does not converge, in the case where there are insufficient samples for the size of the private key, the solving algorithm still constrains the private key to some region of the mod space. Cryptographically there is still value in that partial solution: depending on the percentage of bits that are solved correctly and the size of the key, the rest can be brute-forced to recover the remainder of the key. In a real world setting an attacker may not have unlimited chances to mount the attack, and consequently it would be fruitful to analyze the usefulness of partial solutions.

The original objective of the project was to mount this attack on SGX-SSL which is an Enclave implementation of OpenSSL that is vulnerable to this signing attack [10]. There are a number of complexities with how OpenSSL represents large numbers and performs arithmetic on them, so I resorted to writing my own (EC)DSA signing program in the interest of simplicity and time. The claims made in this paper would be strengthened with a working demonstration of attacking a production library, so I definitely intend to work on that following this class.

This attack uses one attack method, page-faulting, but as discussed in the background there are a number of other ways to attack programs in an enclave context. One could explore attacking DSA signing using single-stepping, and compare the overhead times and accuracy

of recovering the key using that method vs. page-faulting. Single-stepping might even enable recovering the key from the key generation stage. Another attack method is looking at the Branch Table Buffer [7] to examine residue from the enclave execution within the untrusted environment to infer the control flow. A promising follow-on study is doing a survey of these methods to attack (EC)DSA signing in an enclave context, and comparing their accuracies and execution times.

7 Conclusion

This paper investigates how an (EC)DSA key can be extracted from a secure enclave running on an untrusted host OS. The project accomplishes this by page faulting on a certain overflow function in order to establish a bound relating known values m, r, and q with an unknown private key x. To mount the attack an enclave program was created with a hidden private key inside, that is used to sign messages passed into it by a calling application. The calling application is running in a patched operating system with a modified SGX driver to instrument page faults on certain pages which would indicate that a certain bound on x holds for a certain m and r value. The bounds were represented as a lattice basis and solved using the LLL reduction algorithm to recover the private key x with perfect reliability. This marks an advancement from previous works which were only able to demonstrate this attack in a non-enclave context, and use cache timing-based methods that are prone to miscalcuation and mistrails when solving for the private key.

References

[1] Alexandre Adamski. Overview of Intel SGX - Part 1, SGX Internals. Quarkslabs Blog. July 5, 2018. https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html

[2] Claus-Peter Schnorr, M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. September 1991. FCT '91 Proceedings of the 8th International Symposium on Fundamentals of Computation Theory. https://dl.acm.org/citation.cfm?id=740436.

[3] Intel Software Guard Extensions Developer Guide. Revision 1.7. Intel Corporation. 2018. https://download.01.org/intel-sgx/linux-1.7/docs/

[4] Intel Corp. Intel Software Guard Extensions SSL. Github Repository. https://github.com/intel/intel-sgx-ssl

[5] Jo Van Bulck, Frank Piessens, Raoul Strackx. 2017. SGX-Step, A Practical Framework for Precise Enclave Execution Control. In 2nd Workshop on System Software for Trusted Execution (SysTEX 2017). October 28th, 2017. Shanghai, China. https://github.com/jovanbulck/sgx-step/blob/master/systex17.pdf

[6] Keegan Ryan, NCC Group. 2018. Return of the Hidden Number Problem. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2018/rohnp-return-of-the-hidden-number-problem.pdf

[7] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. August 18, 2017. https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-lee-sangho.pdf

[8] Shweta Shinde, Zheng Leong Chua. Preventing Page Faults from Telling Your Secrets. May 30, 2016. https://dl.acm.org/citation.cfm?id=2897885.

[9] Naomi Benger, Joop van de Pol, Nigel P. Smart, Yuval Yarom. Ooh Aah... Just a Little Bit" : A small amount of side channel can go a long way. June 2014. IACR-CHES-2014. https://eprint.iacr.org/2014/161

[10] OpenSSL Software Foundation. Changelog. https://www.openssl.org/news/changelog.html

[11] Victor Costan, Srinivas Devdas. Intel SGX Explained. 2016. MIT CSAIL. https://eprint.iacr.org/2016/086.pdf.

[12] Yuanzhong Xu, Weidong Cui, Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. 2015 IEEE Symposium on Security and Privacy. https://www.ieee-security.org/TC/SP2015/papers-archived/6949a640.pdf

[13] Yuval Yarom, Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. September 2013. University of Adelaide. https://www.usenix.org/node/184416.