CS 741 - Report

Cache Attack on SGX

Team Details:

170050068 - Killari Ramprasad
170050081 - Sailendra Bathi Babu
170050083 - Bonela Mahith
170050100 - Ramya Narayanasamy

1. Abstract:

Intel Software Guard Extensions (SGX) provides a trusted execution environment (TEE) to run code and operate sensitive data. SGX provides runtime hardware protection where both code and data are protected even if the code components are malicious. However, recently many attacks targeting SGX have been identified and introduced that thwart the hardware defense provided by SGX.

In this paper, we demonstrate software-based side-channel attacks from a malicious SGX enclave targeting co-located enclaves. Our attack is malware running on SGX hardware, abusing SGX protection features to conceal itself. We perform a Prime+Probe cache side-channel attack on a co-located SGX enclave running RSA implementation that uses a constant time multiplicative primitive. We extract the RSA key using multiple traces very quickly

Due to the interest of time we were able to only write code for key recovery, which is the last step of the entire attack. Hence, we extended the theoretical aspect of the project to discuss some countermeasures.

2.Introduction:

Modern operating systems isolate user processes from each other to protect secrets in different processes. Cloud providers use virtualization as additional protection using a hypervisor. Hypervisor isolated different tenants that are co-located on the same physical machine but do not protect them against a possibly malicious cloud provider. Although hypervisors provide functional isolation, side-channel attacks are often not considered. Consequently, researchers have demonstrated various side-channel attacks, especially those exploiting the cache. Cache

side-channel attacks can recover cryptographic secrets such as AES and RSA keys, across virtual machine boundaries.

Trusted execution environments (TEEs) create isolated environments where sensitive code can run with a higher security level than the operating system. Intel Software Guard Extensions (SGX) which are Intel's hardware extensions in their CPUs, starting with the Skylake microarchitecture is an example of a TEE. SGX utilizes enclaves to isolate the execution environment from other applications, the operating system's kernel, and the hypervisor. SGX can run arbitrary code on general hardware and is suitable for cloud environments where it isolates running code and data from the untrusted environment. In SGX, the root of trust is based only on the application code itself and the hardware implementation of the CPU. Unfortunately, a relatively large number of flaws and attacks have been published by researchers over the last few years [1].

In this paper, we study a cache attack from within a malicious enclave that is extracting keys from co-located enclaves. The attack monitor cache access patterns of an RSA signature process to recover RSA keys in a semi-synchronous attack as done in [2]. The malware code is completely invisible to the OS and cannot be analyzed due to the isolation provided by SGX.

Section 3 contains some background information on Intel SGX, Cache Attacks and Side Channel Attacks on RSA which is required to understand the attack. Section 4 has details about the threat model, the attack setup and procedure. Section 5 contains the work that we have done and some observations we found interesting about the attack. Section 6 mentions a few countermeasures. Section 7 contains the challenges faced during the process. Section 8 mentions all the references we used in the project. Section 9 contains demo code with.

3.Background:

This section covers topics that help understand the side channel used to retrieve sensitive information. We discuss the basic functionality of Intel SGX, cache attacks, and side-channel attacks on RSA.

3.1 Intel SGX:

Intel SGX protects the execution of user programs in “Enclaves” whose memory region can be only accessed by itself, any other accesses will be blocked by the CPU. As this policy is enforced in the hardware, Enclaves remain safe even if the malware has obtained kernel privileges.

An enclave resides in the virtual memory area of an ordinary application process. The enclave virtual memory region can only be backed by physically protected pages from the Enclave Page Cache (EPC). The EPC is a contiguous physical block of memory in DRAM that is encrypted using hardware encryption, which protects enclaves against hardware attacks.

Loading and creation of Enclaves are done by the OS. To protect the integrity of the enclave code, the loading procedure is measured by the CPU. If the measurement does not match the value specified by the enclave developer, the CPU will refuse to run the enclave. Before the execution of the Enclave, the OS has full access to its binary. Hence it cannot carry any hard-coded secrets and secret information can be given to the Enclave only during run-time. And Enclave malware will attempt to hide from antivirus software by encrypting the malicious payload.

There have been speculations that SGX could be vulnerable to cache side-channel attacks. In fact, Intel does not consider side-channel attacks as part of the SGX threat model and thus states that SGX does not provide any specific mechanisms against side-channel attacks. However, they also explicitly state that SGX features still impair side-channel attacks. Intel recommends using SGX enclaves to protect password managers and cryptographic keys against side channels and advertises this as a feature of SGX.

3.2 Cache Attacks:

There have been many cache-based timing attacks against SGX enclaves published in the literature. Common to all of them are exploitation of cache-hierarchy systems and the fact that the caching of memory loads from DRAM leaves effects in the system state which are measurable from outside the protected application. Cache attacks exploit the timing difference between the CPU cache and the main memory in order to infer information on other processes running in the same system.

What these attacks show is that SGX enclaves are vulnerable to the same cache attacks against secret dependent information processing as any software application. There are a number of different general (non-SGX specific) techniques for extracting information from side channels, some are *Flush+Reload*, *Prime+Probe*, *Evict+Time*, *Evict+Reload* and *Flush+Flush*.

Prime+Probe is a cache attack technique wherein the attacker constantly primes (evicts) the cache set and measures the time taken by this step. The amount of time taken by this Prime step is correlated to the number of ways in which this cache set has been replaced by other programs. This information can be used to derive whether or not a victim process has performed specific secret-dependent memory access.

This attack includes 3 stages(Prime, Victim Access, Probe). In Prime, the attacker fills a portion of the cache with his own dummy data. In Victim Access, the attacker waits for the victim to make particular set accesses in the cache, hoping to see key-dependent cache utilization and

this also results in the eviction of one or more of the attacker's dummy data blocks in the set. In Probe, the attacker performs a per-set timed re-access of the previously primed data, if the probe time is high then the attacker deduces that it(cache set) was accessed by the victim.

The cache timing attacks are based on the difference in memory access times observed by a spy process. On a LAN, the timing of the decryption operation on a web server could reveal information about private keys stored on the server. These attacks are proposed to recover AES, DES, and RSA cryptographic keys. The operation in RSA that involves a private key is the modular exponentiation. $M = C^d \bmod N$ where 'd' is the private key,' C' is ciphertext, 'N' is RSA modulus. For a timing attack, the attacker needs to have the target system compute $C^d \bmod N$ for several carefully selected values of C. By precisely measuring the amount of time required and analyzing the timing variations, the attacker can recover the private key '*d'* one bit at a time until the entire exponent is known.

3.3 Side-Channel Attacks on RSA:

RSA is widely used to create asymmetric signatures , and is implemented virtually by every TLS library, such as OpenSSL or mbedTLS, formerly known as PolarSSL. mbedTLS is used in many well-known open source projects such as cURL and OpenVPN. The small size of mbedTLS is well suitable for the size-constrained enclaves on Intel SGX.

RSA essentially involves modular exponentiation with a private key, where the exponentiation is typically implemented as square-and-multiply as shown in Algorithm 1. The algorithm sequentially scans over all exponent bits. Squaring is done in each step while multiplication is only carried out if the corresponding exponent bit is set. An unprotected implementation of square-and-multiply is vulnerable to a variety of side-channel attacks, in which an attacker learns the exponent by distinguishing the square step from the multiplication step.

Earlier versions of mbedTLS were vulnerable to a timing side-channel attack on RSA. Due to this attack, current versions of mbedTLS implement a constant-time multiplication for RSA. Additionally, instead of using a dedicated square routine, the square operation is carried out using the multiplication routine. Thus there is no leakage from a different square and multiply routine as exploited in previous attacks on square and multiply algorithms. However, the secret-dependent accesses to the buffer containing b still leak the exponent.

Algorithm 1: Square-and-multiply exponentiation

```
Input : base b, exponent e, modulus n
Output: b^e mod n
X ← 1;
for i ← bitlen(e) downto 0 do
        X ← multiply(X , X );
        if ei = 1 then
                X ← multiply(X , b);
        end
end
return X;
```

4.Threat Model and Attack Setup:

In this section we present our model. We explain how the malware can circumvent SGX's isolation guarantees. We explain how to mount a **Prime+Probe attack on an RSA signature** computation running inside a different enclave. In our threat model, both the attacker and the victim are running on the same machine. The machine can either be a user's local computer or a host in the cloud. We focus the discussion about attack when the attacker and victim are running on a local machine.

The figure below shows an overview of our naive setup. The victim runs a cryptographic computation inside the enclave to protect it against any attacks. Both the attacker and the victim use Intel's SGX feature and are therefore subdivided into two parts, the enclave and the loader, i.e main program that instantiated the enclave.

This attack includes 3 stages(Prime, Victim Access, Probe). In **Prime**, the attacker fills a portion of the cache with his own dummy data. In **Victim Access**, the attacker waits for the victim to make particular set accesses in the cache, hoping to see key-dependent cache utilization and this also results in the eviction of one or more of the attacker's dummy data blocks in the set. In **Probe**, the attacker performs a per-set timed re-access of the previously primed data, if the probe time is high then the attacker deduces that it(cache set) was accessed by the victim.

Victim:

The victim is an unprivileged program that uses SGX to protect an RSA signing application from both software and hardware attackers. Both the RSA implementation and the private key reside inside the enclave. Thus they cannot be accessed by system software or malware on the same host. Moreover, information leakage from the enclave should not be possible due to hardware

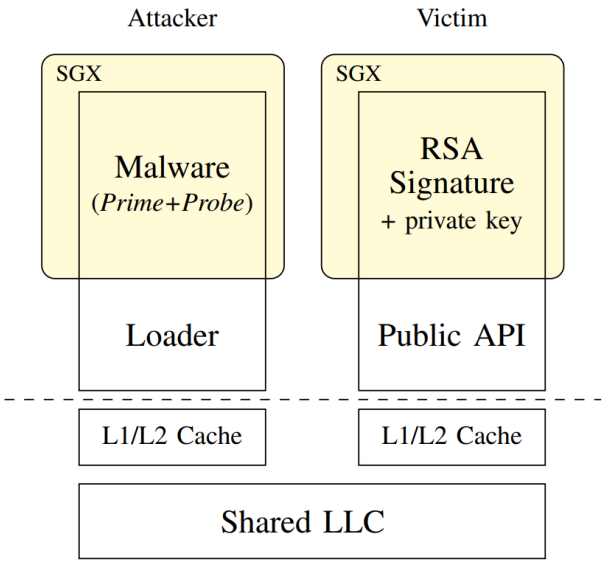
isolation and memory encryption. The victim uses the RSA implementation of the widely deployed mbedTLS library. The victim provides an API to compute a signature for provided data.

Attacker:

The attacker runs an unprivileged program on the same host as that of the victim. The goal of the attacker is to stealthily extract the private key from the victim enclave. Therefore, the attacker uses the API provided by the victim to trigger signature computations. The attacker targets the exponentiation step of the RSA implementation. To perform the exponentiation in RSA, mbedTLS uses a windowed square-and-multiply algorithm in the Montgomery domain.

As explained previously, to prevent information leakage from function calls, mbedTLS uses the same function (mpi_montmul) for both the square and multiply operation (see Algorithm 1). The function takes two parameters that are multiplied together. For square operation the function is called with the current buffer as both arguments. For the multiply operation, the current buffer is multiplied with a buffer holding the multiplier (base b). This buffer is allocated in the calling function using calloc. Due to the deterministic behaviour of the calloc implementation, the used buffers always have the same virtual and physical addresses. Thus the buffers are always in the same cache sets. The attacker can mount a prime+probe attack on the cache sets containing the buffer.

In order to remain stealthy, all parts of the malware that contain attack code reside inside an SGX enclave. Our only assumption on the hardware is that the attacker and the victim run on the same host system. This is the case on both personal computers as well as on co-located Docker instances in the cloud. The last-level cache is shared between all CPU cores.



Extracting Private Key Information:

A successful prime+probe attack requires two primitives: a high resolution timer to distinguish cache hits from misses and a method to generate an eviction set for an arbitrary cache set.

The rdtsc and rdtcp instructions, which read the timestamp counter are usually used for fine grained timing outside enclaves. Inside SGX enclaves, these instructions are not permitted as they might cause a VM exit. Therefore we rely on a different timing source. The idea is to have a dedicated thread incrementing a global variable in an endless loop. This global variable serves directly as a timing source.

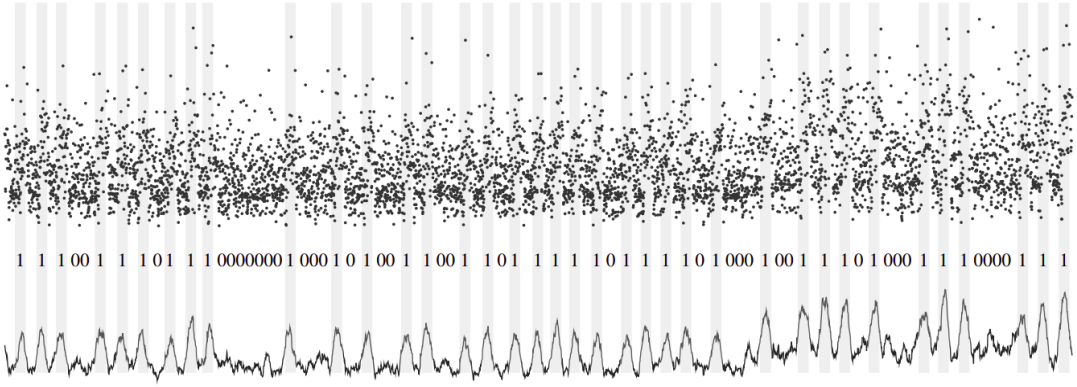
Accessing two virtual addresses that map to the same DRAM bank but a different row is slower than any other combination of virtual addresses. The algorithm for generating eviction sets exploits this DRAM timing differences that are due to DRAM organization.

We perform a Prime+Probe cache side-channel attack on a co-located SGX enclave running an up-to-date RSA implementation that uses a constant time multiplication primitive.In a semi-synchronous attack, we extract 96% of an RSA private key from a single trace. We extract the full RSA private key in an automated attack from 11 traces within 5 minutes

Recovering the private key from the recorded traces of the victim enclave. Key recovery comes in three steps. First, traces are pre-processed. Second, a partial key is extracted from each trace. Third, the partial keys are merged to recover the private key.

High values raw measurement data correspond to cache misses whereas low values indicate cache hits. Timing measurements have a varying sample rate. This is because a cache miss delays the next measurement while cache hits allow more frequent measurements. To simplify the subsequent steps, we convert the measurements to a constant sampling rate.

To automatically extract a partial key from a resampled trace, we first run a peak detection algorithm. We delete duplicate peaks, e.g., peaks where the corresponding RSA multiplications would overlap in time. We also delete peaks that are below a certain adaptive threshold, as they do not correspond to actual multiplications. The trace data when visualized looks like the below figure.



In the final key recovery, we merge multiple partial keys to obtain the full key. We quantify partial key errors using the edit distance. The edit distance between a partial key and the correct key gives the number of bit insertions, deletions and flips necessary to transform the partial key into the correct key.

The full key is recovered bitwise, starting from the most-significant bit. The correct key bit is the result of the majority vote over the corresponding bit in all partial keys. Before proceeding to the next key bit, we correct all wrong partial keys which did not match the recovered key bit according to the edit distance information. This step is done as part of our implementation

5.Our Work:

Our work consisted of first trying to understand the attack procedure and necessary background needed for that. It started by reading about SGX enclaves and then finishing this paper [2]. Once that is done, the next part is implementing the attack. To implement the attack, even partially, understanding of the algorithms mentioned in the paper is necessary. There are a lot of missing functions that are very important to implement the algorithm. Most of the time was spent in trying to figure out what exactly is the complete code flow and the missing helper functions.

We realized that implementation of the full attack would not be possible according to the deadlines. Hence, we focussed on finding a feasible step to code. The offline part is simpler to implement and demonstrate but the data from the online step is necessary. The online part does not require such data, but needs setting up a lot of parameters. The basic step we could do was eviction set generation. Even this requires knowledge on the DRAM configuration, assembly instructions and address mappings and moreover, the same code might not work on all our team's machines due to different configurations.

We went through a few github repositories like [6] hoping that we could get some idea on how to proceed with eviction set generation. But these were not so easy to understand and not SGX specific. Next option was to somehow create our own noisy data to demonstrate the key recovery from partial keys. This involved using a python program to randomly corrupt bits of the binary sequence corresponding to the secret key. This artificial trace data was then used as inputs to the key recovery step. Full key is recovered bitwise, starting from the most-significant bit. The correct key bit is the result of the majority vote over the corresponding bit in all partial keys. Before proceeding to the next key bit, we correct all wrong partial keys which did not match the recovered key bit. We compute the edit distance over a lookahead window of a few bits and perform the necessary actions to transform one key to another.

We ran our algorithm with 1024, 2048 and 4096 bit keys as input. The results are shown in the table below. The noise rate of the partial keys was 4% in all the cases. Algorithm 2 explains the key recovery

Key size	No. of traces	Lookahead size	Time Taken
1024 b	10	10	0.6 seconds
2048 b	15	10	2 seconds
4096 b	20	20	12 seconds

Performance: Key size vs parameters

Algorithm 2: Private Key Recovery

Input: keys: bit strings, lookahead: int
Output: key: bit string

```
key ← [];
i ← 0;
while True do
    keybit ← majority(keys, i);
    if keybit =⊥ then
        return key;
    end
    key[i] ← keybit;
    correct ← {};
    wrong ← {};
    foreach k in keys do
        if k[i] = keybit then
            correct ← correct ∪ k;
        else
            wrong ← wrong ∪ k;
        end
    end
    foreach kw in wrong do
        actions ← {};
        foreach kc in correct do
            actions ← actions∪
              EditDistance(kw[i : i + lookahead],
                           kc[i : i + lookahead]);
        end
        ai ← 0;
        while kw[i] ≠ keybit do
            action ← majority(actions, ai);
            apply action to kw[i];
            ai++;
        end
    end
    i++;
end
```

```
function majority(set, idx) begin
    counter[] ← 0;
    foreach array in set do
        element ← array[idx];
        increment counter[element];
    end
    return element with max. counter;
end
```

We have observed some interesting problems that came up in performing the attack and noted how the authors of the paper we referred to solved them.

Interesting Observations:

1. SGX enclaves do not share memory with other enclaves, the operating system or other processes. Thus, *Flush+Reload* attacks on SGX are not possible because *Flush+Reload* attacks rely on attacker and victim sharing memory (i.e a shared library or page deduplication). The attacker flushes a shared memory line from the cache to then measure the time it takes to reload the cache line. This then reveals whether or not the victim accessed this exact cache line.
2. Most cache side channel attacks use rdtsc or rdtscp instructions which read the timestamp counter for their implementations. But these instructions are not permitted inside the SGX enclave as they might cause a VM exit, which compromises the security of the enclave. Hence, in this attack a new timing mechanism is created. A global variable is used as the timing source and a separate counter thread is employed to increment the global variable. This method supposedly takes 0.87 cycles per increment as opposed to 1 cycle per increment of the rdtsc instruction. Thus, providing a more fine-grained timing source.
3. Alternately accessing two virtual addresses that map to the same DRAM bank but a different row is significantly slower than any other combination of virtual addresses. The algorithm for generating eviction sets exploits this DRAM timing differences that are due to DRAM organization.
4. The online phase should be as efficient as possible. Once, we have the data from the online phase, it is okay to spend time processing the data in the offline phase. For this, we need to keep the processing in the online phase to a minimum, which helps in achieving high sampling rate for our measurements. To achieve this, we only take the timestamps of the cache misses in our measurements.
5. In the offline phase, instead of recovering the RSA key from combining the traces (traces contain cache hit and miss data using timestamps), the attack first recovers multiple partial keys from individual traces and then recovers the final key using these partial keys. This is because the traces are affected by multiple noise sources and thus, making the trace alignment difficult. This approach has much less computational overhead than trace alignment since key recovery is performed on partial keys of length 4KB instead of full traces containing several thousand measurements.

Since, we were not able to do much regarding the implementation part, we decided to extend the project and discuss some countermeasures.

6.Countermeasures:

The countermeasures can be classified into 3 categories based on the modification required

A.Source level:

1. Exponent Binding : Signing the message as m ^ (d+k·φ(N)) where k is a random binding number. This method only works when single trace recovery isn't possible and relies on the presence of a random number source.
2. Bit Slicing : The main idea is to use only bit operations for computations throughout the algorithm. No lookup tables or branches are used in these algorithms and thus, they are not vulnerable to cache attacks. However this is restricted to only a few algorithms.

B. OS level (assuming the OS is benign):

1. Eliminating Timers : Removing access to high-resolution timers and all forms of simultaneous multithreading effectively eliminate access to sufficiently accurate timers and mitigate many attacks
2. Enclave Coloring : Here the cache is partitioned into multiple smaller parts. Each of the parts spans over multiple cache sets, and no cache set is included in more than one part. However this requires trusting the OS, which is against the core philosophy of SGX

C. Hardware Level:

1. Combining Intel CAT with SGX : Any cache sharing between SGX enclaves and the outside as well as co-located enclaves could be eliminated. It would protect co-located enclaves as well as the operating system and user programs against malicious enclaves.

2. Secure RAM : An additional secure memory element that resides inside the CPU with data stored within not cacheable. SGX driver then provides a special API to acquire this element for temporarily storing sensitive data. Providing such a secure memory element per CPU core would even allow parallel execution of multiple enclaves. As data from this element is only accessed by one program and is never cached, cache attacks and DRAM-based attacks are not possible anymore.

7.Trails and Tribulations:

1. This attack involves many intricate details on various topics like assembly instructions and DRAM configuration knowledge. Hence, getting a basic understanding of the attack procedure took a lot more time than expected.

2. Algorithms mentioned in the research paper skips out on a few helper functions which are considered trivial by the authors. Hence, we had to understand what the missing functions were doing to get a clear understanding of the attack.
3. Once understanding the theoretical aspect of the project was done, figuring out the implementation proved to be very challenging. First of all, running code on SGX enclaves was not as straightforward as we expected. Then we hit a roadblock regarding conversion of the given algorithms into working code.
4. We referred to a lot of git repositories regarding the exact function calls for eviction, timing thresholds etc. But they were not much useful for us as they were not SGX related and use few instructions which are not compatible with SGX.
5. Keeping track of what works and what does not work among the team was difficult because different members of the team have different machine configurations and hence the code arguments and parameters need to be configured differently.
6. Since the attack is very complicated, we have decided not to implement a full attack and focus on parts which are feasible before the deadline. Offline part of the attack is not so complicated as that of the online part. But requires data from the online part to be provided as inputs. For this we e-mailed two of the authors asking if they are willing to provide the trace data for the project. Thankfully, one of the authors replied with the trace files [7] . But we did not have enough time to go through the data as the reply arrived on the day of project submission(i.e 19th April 2021). We had already started working on creating artificial trace data for the sake of demo.

8.References:

1. **A Survey of Published Attacks on Intel SGX**
2. **Malware Guard Extension: Using SGX to conceal cache attacks**
3. **Intel SGX explained**
4. **CacheZoom: How SGZ amplifies the power of cache attacks**
5. **Timing Side-Channel attack on RSA**
6. **Finding Eviction Sets**
7. **Trace data obtained from Online Step**

9.Appendix

Code for *generateData.py*:

Program to generate the trace data.

```
from Crypto.Util import number
import random

# parameters
NUM_BITS = 1024  # key size in bits
NUM_TRACES = 10  # number of traces to generate

fractionNoise = 0.04    # fraction of bits in each trace to corrupt

# name of the file to write data into
datafile = "data.txt"

numNoise = int(fractionNoise*NUM_BITS)

# generate a random prime number(RSA key) of length NUM_BITS
keyDecimal = number.getPrime(NUM_BITS)

keyBinary = bin(keyDecimal)[2:]

# function for creating a trace given a binary string as input.
def createTrace(binaryStr):
   binaryList = list(binaryStr)

   # sample the indices to corrupt
   idx_list = random.sample(range(0,NUM_BITS), numNoise)

   for idx in idx_list:
       # choose between bit flip, insertion and deletion at a chosen
index(idx)
       random_coice = random.choice([0, 1, 2])
       if(random_coice == 0):
           # flip bits
           try:
               if(binaryList[idx] == '0'):
                   binaryList[idx] == '1'
               else:
                   binaryList[idx] = '0'
           except:
               pass
```

```
        elif(random_coice == 1):
            # insert bits
            rand_ele = random.choice(['0', '1'])
            try:
                binaryList.insert(idx, rand_ele)
            except:
                pass
        else:
            # delete bits
            try:
                binaryList.pop(idx)
            except:
                pass

    return "".join(binaryList)


# write the original key, followed by traces into the data file.
fp = open(datafile, 'w+')

fp.write(keyBinary+'\n')

for _ in range(NUM_TRACES):
    trace = createTrace(keyBinary)
    fp.write(trace + '\n')

fp.close()
```

Code for *keyRecovery.py*:

Program to recover the secret key from generated trace data

```
from collections import defaultdict
import sys

# lookahead window size
lookahead = None
lookahead = int(sys.argv[1])
```

```
###################################
# function to backtrack the DP array to get edit actions
def editDistance(s1, s2, dp):
    i = len(s1)
    j = len(s2)
    edits = []

    # Check till the end
    while(i > 0 and j > 0):
        # If characters are same
        if s1[i - 1] == s2[j - 1]:
            edits.append("NA")
            i -= 1
            j -= 1

        # Replace
        elif dp[i][j] == dp[i - 1][j - 1] + 1:
            edits.append("r"+str(s2[j-1]))
            j -= 1
            i -= 1

        # Delete
        elif dp[i][j] == dp[i - 1][j] + 1:
            edits.append("d")
            i -= 1

        # Insert
        elif dp[i][j] == dp[i][j - 1] + 1:
            edits.append("i" + str(s2[j-1]))
            j -= 1

    while(j>0):
        edits.append("i" + str(s2[j-1]))
        j -= 1

    while(i>0):
        edits.append("d")
        i -= 1
```

```
    edits.reverse()
    return edits

# Function to compute the DP matrix for Edit distance
def editDP(s1, s2, flag):
    # edits = ["NA"]*(len(s1)+1)
    len1 = len(s1)
    len2 = len(s2)
    dp = [[0 for i in range(len2 + 1)]
             for j in range(len1 + 1)]

    # Initialize by the maximum edits possible
    for i in range(len1 + 1):
        dp[i][0] = i
    for j in range(len2 + 1):
        dp[0][j] = j

    # Compute the DP Matrix
    for i in range(1, len1 + 1):
        for j in range(1, len2 + 1):

            # If the characters are same
            # no changes required
            if s2[j - 1] == s1[i - 1]:
                dp[i][j] = dp[i - 1][j - 1]

            # Minimum of three operations possible
            else:
                dp[i][j] = 1 + min(dp[i][j - 1],
                                   dp[i - 1][j - 1],
                                   dp[i - 1][j])

    if(flag):
        edits = editDistance(s1, s2, dp)
        return edits, dp[len1][len2]
    else:
        return None, dp[len1][len2]
####################################

####################################
```

```
# majority function to check which what is the majority element of all the
given list of lists
# for a given particular index
def majority(st,idx):
    counter = defaultdict(int)
    for l in st:
        try:
            counter[l[idx]] += 1
        except:
            counter["NAE"] += 1
    return max(counter,key=counter.get)
####################################

####################################
# code to read the traces from a file
keys = []
with open("data.txt") as f:
    lst = f.readlines()
    original_key = lst[0][:-1]
    ll = lst[1:]
    for s in ll:
        # print(editDP(s[:-1], original_key, False))
        keys.append(list(s)[:-1])
####################################


key = [] #key is the final output key
i=0

while(True):
    keybit = majority(keys,i) # majority bit in all the traces for i index
    if(keybit=="NAE"):
        break

    key.append(keybit)
    correct = []
    wrong = []

    ####################################
    #This block places all the traces with the majority bit in correct
```

```
    #and the rest in wrong
    for idx,k in enumerate(keys):
        if(i<len(k) and k[i]==keybit):
            correct.append(idx)
        else:
            wrong.append(idx)
    ###################################

    for kw in wrong:
        actions = [] #actions is a list of lists
        for kc in correct:

actions.append(editDP(keys[kw][i:i+lookahead],keys[kc][i:i+lookahead],
True)[0])
            #editDP returns list which says which action(replace,delete or
insert) to be applied on which index of a wrong trace
        ai=0
        no_of_deletes=0 #these are just helper variables
        no_of_inserts=0
        while(i<len(keys[kw]) and keys[kw][i]!=keybit):
            action = majority(actions,ai) #action is the majority of all
the lists in actions for a particular index
            if(action == "NAE"):
                break

            # apply action to the index of a wrong trace
            if(action=="d"):
                keys[kw].pop(i+ai-no_of_deletes)

                no_of_deletes += 1

            elif(action[0]=="i"):
                keys[kw].insert(i+ai-no_of_deletes, action[1])
                no_of_inserts += 1

            elif(action[0]=="r"):
                keys[kw][i+ai-no_of_deletes] = action[1]

            ai+=1
```

```
   i+=1


final_key = "".join(key)
# checking if the recovered key matches with the original key
print("edit distance between final key and original key:",
editDP(final_key, original_key, False)[1])
print("Key recovered:", final_key == original_key)
```