CS 1657 Project 1

G. Constantine, G. Heinrichs, C. Simons

24 February 2023

1 Introduction (W0-W1)

For the purposes of this project, we define problem P to be the problem of **modular exponentiation**. Modular exponentiation is defined as, for given $y \in \mathbb{Z}$, $x \in \mathbb{Z}$, and $n \in \mathbb{Z}^+$,

$$y^x \bmod n$$

In number theory, this is useful for working with groups, and limiting the results $r \mid 0 \leq r \leq n-1$ of numerical operations.[1] Because many cryptographic operations rely on cyclic groups, modular exponentiation is a core aspect of some major cryptosystems. Two major cryptographic methods rely heavily on modular exponentiation to function: Rivest-Shamir-Adleman (RSA) encryption, and classic Diffie-Hellman (DH) encryption. RSA key systems are composed of the following numbers:

- some large primes p and q
- $n = pq$
- $\phi(n) = (p-1)(q-1)$
- $e \in \mathbb{Z} \mid \gcd(e,\ \phi(n)) = 1$
- $d \in \mathbb{Z} \mid ed \equiv 1 \bmod \phi(n)$

Of these numbers, RSA keypairs are composed of a private key $(p,\ q,\ d)$ and a public key $(n,\ e)$. Then, to encrypt a message M, a user computes

$$M^e \bmod n$$

To decrypt a ciphertext C a user computes

$$C^d \bmod n$$

Already, one can see that modular exponentiation with large exponents and modular bases is the core of RSA's encryption scheme.

Diffie-Hellman uses a different scheme, allowing users to share a secret over an entirely public channel. In the DH crypto scheme, users agree upon a finite cyclic group G of order q where q is prime, and a generator g. Then, each user selects a random number, which we will call $a \in \{1, ..., q-1\}$ and $b \in \{1, ..., q-1\}$ respectively. After this, each user calculates $g^x \bmod q$ where x is their random number, and sends this value to the other, who raises it to their own random number, resulting in each user leaving the system with $g^{ab} \bmod n$. As with RSA, the basis of the entire cryptosystem lies on modular exponentiation. Combined, these two encryption schemes form the basis of modern public key cryptography, making modular exponentiation absolutely pervasive in modern security. In this paper, we will focus on modular exponentiation in relation to the RSA scheme.

[1] [1] Lynn.

Through completion of our project, we set out to accomplish several goals. First and foremost, we wanted to gain a greater and holistic understanding of timing attacks. This involves implementing several algorithms, such as modular exponentiation, Granlund's Algorithm 6, and blinding, which are important in several cryptographic algorithms mentioned above. Furthermore, this involves analyzing these algorithms as a whole and by their parts, such as various steps that may prove to lead to vulnerabilities in a program. The analysis portion of our goals includes several timing and statistical tests, with graphs included for better understanding and visualization. Through the specified implementation and analysis, we hoped to gain better insight into what makes these algorithms vulnerable, and best practices for eradicate or partially resolve these issues.

2 Algorithm A (C2)

Given the ubiquity of modular exponentiation in modern cryptography, much work has been done to create optimal algorithms to compute modular exponents. Before the advent of timing attacks, the goal for designing such algorithms was generally time-efficiency, since operations on the huge integers that underpin encryption are generally very slow. One simple and relatively fast algorithm for computing modular exponentiation is known as *successive squaring.* Successive squaring is the basis for our algorithm A, which we define as follows (in pseudocode):

```
  Input: $C \in \mathbb{Z},\ d \in \mathbb{Z},\ N \in \mathbb{Z}^+$
  Output: $result = m^e \bmod N$
1 $result \leftarrow 1$
  // length() gets the length in bits of the input
2 $b \leftarrow \text{length}(d)$
3 for $k \leftarrow 0$ to $b$ by $1$ do
      // Square the current result unconditionally
4     $result \leftarrow result^2 \bmod N$
      // Extract d$_k$
5     $d_k \leftarrow d >> (b - k - 1)\ \&\ 1$
6     if $d_k = 1$ then
          // Multiply result by input if this bit is set
7         $result \leftarrow result \times C \bmod N$
8     end
9 end
```

Algorithm A: Modular exponentiation via successive squaring. C source code for this algorithm is available at `/src/modexp.c`. Note that we have condensed what may be several lines of code or library function calls into what is, to the best of our knowledge, their ultimate effects, for clarity.

This algorithm makes use of *conditional execution.* On line 6, the algorithm checks whether the current bit is set while iterating through each bit of the exponent. If and only if that bit is set, the following multiplication and modular division on line 7 will take place. This corresponds to a *branch condition* instruction in the generated assembly code from compiling this algorithm, which will cause the instructions on line 7 not to execute at all if the bit is not set. If efficiency is the developer's only concern, this is great, because it implies that some instructions can be forgone entirely if they are not necessary, and the algorithm will execute only *exactly* what is necessary to calculate its output.

However, this is a double-edged sword. Because of the time that it takes to complete the multiplication and modular division operations, the variability in the number of modular divisions has significant implications for side-channel timing attacks. If an exponent d contains many set 1 bits, then many more multiplications will occur than for an exponent with less bits set. This leads to the possibility of predicting a private exponent d bit-by-bit by carefully observing running time for the modular exponentiation step. In testing our algorithms, we counted mod operations performed in each input scenario. For algorithm A, we made the assumption that the modular function being used would optimize out inexpensive mods (mod operations where the operand is less than N already). Therefore, we only counted mods in Algorithm A if the operand was greater than or equal to N, whereas in our proposed Algorithm B (discussed below) we assumed that a proper implementation would not do such optimizations, and each mod is performed regardless. We found that Algorithm A did indeed experience a variable number of mod operations linearly proportional to key size, and that even with a fixed key but different messages, mods still varied in a sort of staircase pattern. This leads us to believe that it would be possible to perform an attack by detecting variance in the actual running time and the running time with guessed bits. We discuss this attack in more detail below.

2.1 Algorithm A Attack Outline(C3-W4)

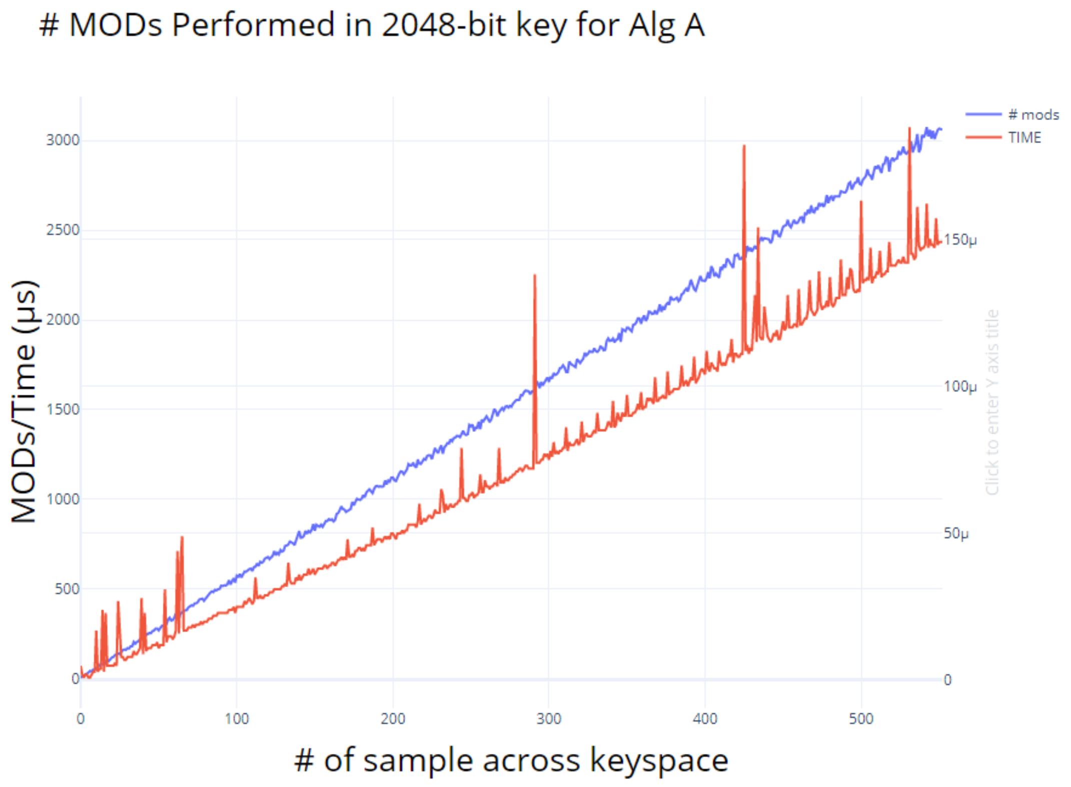

Figure 1: Mod operations over increasing key sizes with a fixed message. We supply these stats to show that mod operations are proportional to key size in Algorithm A.

The statistics used for this graph were generated on a single message $C \in \mathbb{N}$. We test the number of mods performed by our Algorithm A when calculating $C^d \bmod N$ for $d \in \{0,1\}^{2048}$. Note that the samples are collected by taking exponentially higher d at every sample, meaning this graph is in logarithmic scale with respect to d. We can see that there is a clear correlation between the number of modulo's performed and d as the line graphed in blue trends upward.

Meanwhile, the red line representing running time in micro-seconds for each key tends to spike. This is a practical concern when measuring fast operations. In practice, an attacker could send many messages per key to find an average time. We dubbed this ”normalizing” the time through averaging the total time after many attempts. This improved the quality of the data, as shown in Figure 2. It is visible from the graph that there is a consistent proportional relationship between the number of mods and total running time, meaning that by *only* timing the algorithm, an attacker would yield information about bits in the key. The following is an outline of how an attacker would obtain information about a key based on the (estimated) number of mods.

Say we wanted to recover bit d_1 (Bit adjacent to MSB). Note that we can assume without loss of generality that d starts with 1, as leading 0's will have no effect on the output. Recall the rules of successive square and multiply: we start with our result as 1 and square at every step, but both square and multiply by C when $d_k == 1$. Consider the following example:

$$\text{if d=00..01}\mathbf{0}\text{0...}\quad \begin{array}{ll} (1)^{2^{2\cdots}}\cdot C^1 & = C^1 \\ \left(C^1\right)^2 & = C^2 \\ \left(C^2\right)^2 & = C^4 \\ \ldots & \ldots \end{array} \qquad \text{if d=00..01}\mathbf{1}\text{0...}\quad \begin{array}{ll} (1)^{2^{2\cdots}}\cdot C^1 & = C^1 \\ \left(C^1\right)^2\cdot C^1 & = C^3 \\ \left(C^3\right)^2 & = C^6 \\ \ldots & \ldots \end{array}$$

Recall that N is typically public in modular exponentiation cryptographic schemes. Let us select two messages Y and Z such that $Y^3 < N$ and $Z^2 < N < Z^3$. We notice that while computing Y^d for *any* d,

$$\text{if d=00..01}\mathbf{0}\text{...}\quad \begin{array}{ll} (1)^{2^{2\cdots}}\cdot Y^1 & = Y^1 \\ \left(Y^1\right)^2 & = Y^2 \\ \ldots & \ldots \end{array} \qquad \text{if d=00..01}\mathbf{1}\text{...}\quad \begin{array}{ll} (1)^{2^{2\cdots}}\cdot Y^1 & = Y^1 \\ \left(Y^1\right)^2\cdot Y^1 & = Y^3 \\ \ldots & \ldots \end{array}$$

Note that no matter what the first two digits of d are, no mods are performed. While calculating Z^d, we yield

$$\text{if d=00..01}\mathbf{0}\text{...}\quad \begin{array}{ll} (1)^{2^{2\cdots}}\cdot Z^1 & = Z^1 \\ \left(Z^1\right)^2 & = Z^2 \\ \ldots & \ldots \end{array} \qquad \text{if d=00..01}\mathbf{1}\text{...}\quad \begin{array}{ll} (1)^{2^{2\cdots}}\cdot Z^1 & = Z^1 \\ \left(Z^1\right)^2\cdot Z^1 & = Z^3[\text{mod}] \\ \ldots & \ldots \end{array}$$

There is an extra mod performed if and only if d_1 is 1, because Z^3 is reached earlier. In practice we would select many messages Y_i and Z_i and compare their runtimes (now that we know that runtime is proportional to the number of mods performed). If we notice that all Z_i tend to take longer than Y_i we recover that $d_1 = 1$. Note that we base this attack on methods discussed by Wing H. Wong[2]. This process can be repeated by selecting messages to target each bit– eventually recovering the key.

[2] [2] Wong.

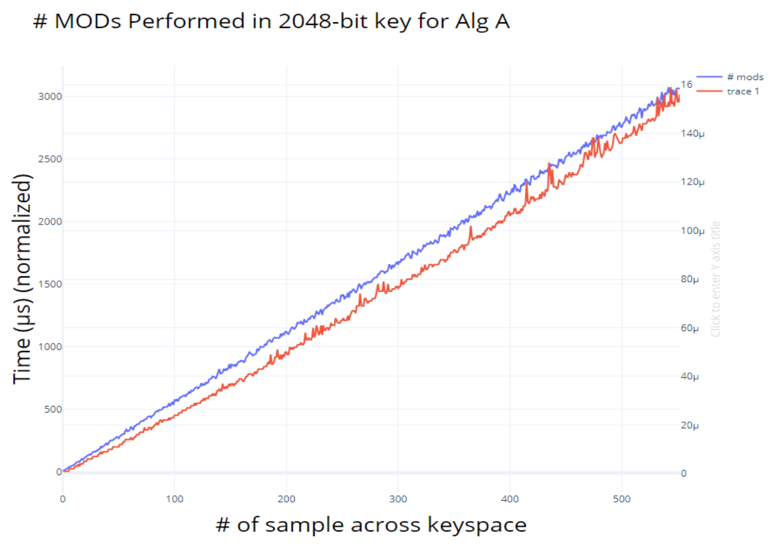

Figure 2: Here we can see more consistent timing information with each time value being the average of 3000 exponentiations per key. Note that there is a slight curvature, and we speculate this may be due to caching after repetitive computation.

3 Algorithm B (C5)

As timing attacks on RSA implementations were originally outlined by Paul Kocher[3], Kocher himself suggested a mechanism for mitigating the information leaked by observing timing, called *blinding*. The idea behind this is to generate a pair of fresh, secret integers (u, v) such that u^d is the multiplicative inverse of $v \bmod n$. Then, multiplying the input message to modular exponentiation by u randomizes the message, and this process can be undone by multiplying the result by v. Then, because the message can no longer be specially crafted to observe variance in timing with different messages, timing information collected is not so useful. Kocher explains that applying this technique, attackers can gain timing distributions for exponentiation, but these are generally normal, and an attacker could not extract exponents from these distributions.[4] However, he continues that a specially designed algorithm for exponentiation could cause spikes to occur with set exponent bits[5], and so we decided to research mechanisms designed to reduce timing variance altogether.

Wherein the primary vector of a timing based side-channel attack on Algorithm A is rooted in the conditional execution of time-consuming mathematical operations, it follows that minimizing or eliminating conditional execution would reduce the likelihood of a timing attack succeeding. Torbjörn Granlund outlined an algorithm to silence as much timing variance in arithmetic operations as possible by eliminating conditional execution and data dependent control flow.[6] To do so, Granlund essentially decomposes the exponent d by creating a pre-computed[7] table of powers of the message m up to a power k of 2. Then, a k-ary slide over the exponent occurs using up to k bits at a time to index into the table and multiply in the cached result. This indexing is done with a mask, so that the entire table is iterated at every step, to silence cache-based attacks.

Our proposed Algorithm B combines both of these mechanisms, which have been shown on their own to be beneficial, to investigate if further benefit can be gained from combining them. Specifically, we do our

[3] [3] Kocher.
[4] Ibid.
[5] Ibid.
[6] [4] Granlund.
[7] In our implementation, we include this pre-computation in timing.

best to implement Granlund's "Algorithm 6" from the aforementioned paper[8], as it is listed, and then apply Kocher's blinding to the input message of this algorithm. Because our implementation is combining these two extant algorithms (which have been outlined in their respective papers), we refrain from listing pseudocode for the entire algorithm, and will now instead analyze the core parts making the algorithm effective.

3.1 Algorithm B Code Analysis

3.1.1 Granlund's Algorithm Implementation

The first iteration of our Algorithm B, we attempted to implement Algorithm 6 out of Granlund's paper, based on the pseudocode he provided. The basics of this algorithm are as follows:

1. Precompute a table of powers of m up until $2^k - 1$ for some $k \geq 1$. In our implementation, k is macro-defined as 6.
2. Loop down from the most significant bit of d to the least significant bit of d in sliding chunks b of up to k bits.
   (a) Square the result (initialized to 1) for as many bits as this iteration will be sliding over.
   (b) Iterate through the table creating a bit mask based on the index at each step, using the bit mask to apply a multiplication of the table entry at index b.

Step 1. is to precompute the table of powers. We define k to be 6, which allows us to precompute $m^0..m^{63}$ as follows:

```
  **Input:** $m \in \mathbb{Z},\ N > 0,\ k \geq 1$
  **Output:** $table$ of length $2^k - 1$
1 $table[0] \leftarrow 1$
2 **for** $i \leftarrow 1$ **to** $2^k - 1$ **do**
3     $table[i] = (table[i-1] \times m) \bmod N$
4 **end**
```

Step 1: Precomputing the powers of m. The corresponding C code can be found at the top of the `granlunds` function in `modexp_silenced.c`

After this step, we proceed to step 2. extracting bits j through i where i is the max of $j - k + 1$ and 0, in case the bottom of the exponent d was reached.[9] We achieve this step with simple shifting and bit masking, however in our implementation shifting actually takes place via truncative division by a power of 2, as libgmp provides no shift function. In step 2.(a), we simply square the result $j - i + 1$, or, usually, k times.

```
  **Input:** $j \leq \text{length}(d),\ 0 \leq i < j$
  **Output:** $result^{2^{j-i+1}}$ (essentially $result^{2^k}$)
1 **for** $s \leftarrow 1$ **to** $j - i + 1$ **do**
2     $result = (result \times result) \bmod N$
3 **end**
  // We believed that reducing s here was necessary, but further inspection
  // shows it will be zeroed later anyway.  Still, this is present in our code.
4 $s = s - 1$
```

Step 2.(a): Unconditionally squaring for the size of the current window. This is the first for loop inside the large while loop in our `granlunds` function.

[8][4] Granlund, 7.

[9]The manner in which we calculate MAX is actually done without the use of any conditionals, by abusing C types, via a mechanism we found in a Techie Delight blog article [5], listed in references.

Step 2.(b) aims to "extract [the] table-entry in a side channel silent way."[10] It does this by iterating over the entirety of the table, then creating a mask that is fully set bits only when the index in the table matches the value of the window of k bits of d being observed at this iteration. By applying this mask and its 1's complement unconditionally, every single step of the iteration does the same arithmetic, but s turns out to be the table entry only when the right index is reached, so there is no short-circuiting or variance in loop running time.

```
  Input: $table,\ s = j - i + 1,\ b = d_j..d_i$
  Output: $s = table[b],\ result = result \times s \bmod N$
1 for $ix \leftarrow 0$ to $2^k - 1$ do
2     $mask \leftarrow \text{eqmakemask}(ix,\ b)$
3     $s \leftarrow (s\ \&\ \sim mask)\ |\ (table[ix]\ \&\ mask)$
4 end
5 $result \leftarrow result \times s \bmod N$
  // Set $j$ down to slide on in next iteration
6 $j = i - 1$
```

Step 2.(b): Applying the table entry multiplication. This is implemented in the second `for` loop inside the large while loop in the `granlunds` function.

The mask creation in step 2.(b) is designed to work without any conditional behavior, preventing a data flow dependency from forming even when indexing into the table, and the application of the mask acts like a conditional operator by collecting a mask of all 1s if the index is equal to b and a mask of all zeroes otherwise, then applying them over a bitwise OR to an interim result variable and the table value. Below is the pseudocode for the creation of the mask:

```
  Input: $a \in \mathbb{Z},\ b \in \mathbb{Z},\ \beta = 2^{4096} - 1$
  Output: $result = 0..0^{\text{length}(a)}$ IF $a \neq b$ ELSE $1..1^{\text{length}(a)}$
1 $result \leftarrow 0$
2 $l \leftarrow \text{length}(a)$ // length here is the bitwise length
3
4 for $i \leftarrow 0$ to $l - 1$ do
5     $result = result\ |\ (((a >> i)\ \&\ 1)\ \hat{}\ ((b >> i)\ \&\ 1))$
6 end
  // Our beta is most likely not the intended beta according to the paper
  // In testing, we were able to make the algorithm work
  // by using a Beta that is at least as large as essentially
  // any table entry would be, and so ours
  // is defined to be $2^{4096} - 1$
  // This is because the bit-length of the output mask
  // is proportional to the mod base beta, and so
  // we wanted to ensure that even a 4096-bit entry
  // would be fully masked out.
7 $result \leftarrow result \bmod \beta$
```

`eqmakemask` **function**: Mask creation used in Granlund's algorithm 6, representing our `eqmakemask` function in `modexp_silenced.c`. Note that in our implementation, β is hard-coded to $2^{4096} - 1$. Thus, we don't do Montgomery reductions to decompose m into base β as does Granlund.

[10] [4] Granlund, 7.

Granlund proves in his paper that this algorithm, if implemented perfectly, will eliminate conditional execution entirely, resulting in constant execution time for all input sizes. Our findings reflect this in our timing results. In addition, we emulated counting mod operations for both algorithm A and algorithm B. We are not under the impression that the library we use for arbitrary precision arithmetic (the GNU Multiple Precision Arithmetic library) optimizes out mods, as it simply performs truncative division and captures the remainder under the hood. Therefore, as discussed above, we did not simulate any optimization in algorithm B, and counted a constant number of mod operations for each input. A more robust implementation would ensure that optimization is not being performed down to the instruction level, however we were unable to make this assurance within the time frame provided. Still, our `granlunds` function (excluding the internals of function calls) used zero `if` statements, and we believe the only `branch` instructions present in the assembly output (found in `/src/modexp_silenced.s`) are those of loop conditions, which are the same for each input.

3.2 Kocher's Blinding Implementation

Our implementation of Kocher's blinding technique is relatively inefficient, but will only be performed once, as we do not repeat inputs. Therefore, we provide no update function. Instead, we generate the pair $(u,\ v)$ by first generating a v at random until it is relatively prime to N, and then generate a corresponding u using the GMP library's native function for calculating multiplicative inverses. Our implementation (found in the `blinding` function in `modexp_silenced.c`) for generating $(u,\ v)$ looks generally as follows (using libgmp types and functions): Then, to apply these primitives, we wrapped a call to Granlund's algorithm outlined

```
  Input: $m \in \mathbb{Z},\ d = e^{-1} \bmod \phi(N),\ e = 65537,\ N > 0,\ B \in \{2048, 4096\}$
  Output: $u,\ v \in \mathbb{Z}$
1 $v \leftarrow$ random $B$ bytes
2 while $gcd(v,\ N) \neq 1$ do
3     $v \leftarrow$ random $B$ bytes
4 end
5 $v\_inverse \leftarrow v^{-1} \bmod N$ // Calculated using GMP function
6 $u \leftarrow v\_inverse^{e} \bmod N$
```

Blinding: Generating u and v for Kocher's blinding. In generating a random v, we use Linux' in-built `getrandom` function, which samples `/dev/urandom`.

above in a function which would multiply the input message by a freshly generated, secret u, then multiply the result of Granlund's by the corresponding v:

```
  Input: $m \in \mathbb{Z},\ d = e^{-1} \bmod \phi(N),\ e = 65537,\ N > 0$
  Output: $u,\ v \in \mathbb{Z}$
1 $u, v \leftarrow$ generate_uv()
2 $m \leftarrow m \times u$
3 $result \leftarrow$granlunds() // Applies above algorithm
4 $result \leftarrow result \times v \bmod N$ // Requires one extra mod operation
```

Applying u and v: This step requires an extra mod operation, and slows down calls to modexp since it generates entirely new u and v each call. For the purposes of educational data collection, we find this acceptable.

3.3 Results of Algorithm B (C6-W7)

Our testing of Algorithm B happened in two stages:

1. Testing only Granlund's algorithm

2. Testing Granlund's algorithm wrapped in Kocher's blinding

In the first stage of this testing, we found the Granlund's algorithm resulted in constant mod operations across inputs (message and exponent) of different sizes, with a visible slowdown in operation time.

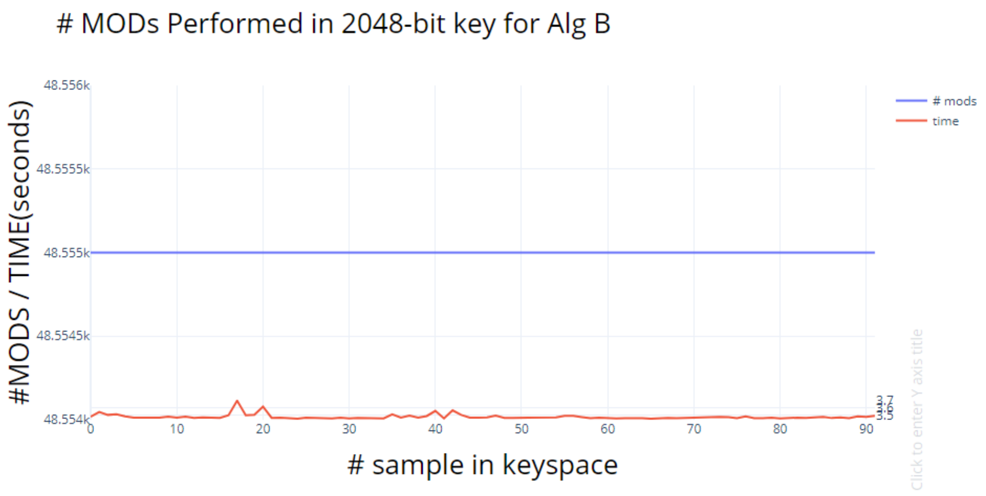

Figure 3: Early results for Algorithm B when it only incorporated Granlund's Algorithm 6. Number of mods remained constant across increasing key sizes up to 2048-bit keys, as expected, since the control flow path should be the same. Note that time fluctuates slightly, probably due to low-level optimizations, but hovers around 3.5 seconds.

Relatively stable running time and constant mod operations as key size increased (given simulated counting) is exactly the outcome that was expected if our implementation was true to the algorithm proposed in Granlund's paper. The intention of Granlund's algorithm 6 is to eliminate conditional execution and ensure that no matter the input's size or bit population count, the data flows through the same path of execution. This stands in stark contrast to Algorithm A, where key size was directly proportional to number of mod operations counted. While lower-level implementation abstractions in the GMP library or even other processes running on the test system could be to blame for the existent variation in timing, the relative stability across sizes shows that even our initial implementation is most likely considerably true to the original algorithm, and is more resistant to timing attacks than the proposed Algorithm A.

In the second stage of testing, we moved on to testing our full Algorithm B: Kocher's blinding on top of Granlund's algorithm 6.
When increasing the key size for our full algorithm B (*Figure 4*), including kocher's blinding, we still don't see a change in mod operations, and timing variation, while more pronounced, is relatively small. We expect that the increase in timing variation is because kocher's blinding introduces extra, semi-random multiplications. Still, the variance is very low, and the number of mod operations is entirely invariant.

In our next test run, we decided to vary message size but fix key, to further investigate if any of the inputs to the algorithm caused variation (*Figure 5*). Mod operations still remained constant across the increasing message size, although a large jump in time (from < 4 seconds to ≈ 4.5) is seen when the message reaches approximately 1500 bits in length. Since mod operations at that step do not change, we hypothesize that this variation could genuinely be because of another process on the test machine (which was not running sterile, without other background processes). Another process could have caused the scheduler to pre-empt the test program and throw timing off, especially given that the timing then returns to previous behavior (until the message exceeds the key size) but at a higher y offset.

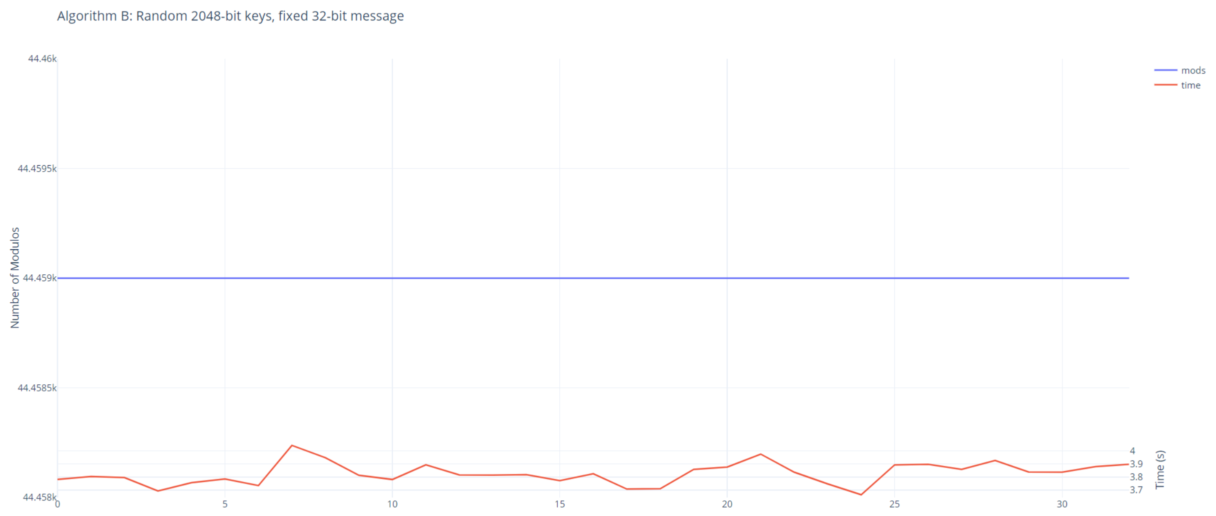

Figure 4: Increasing the key size over a fixed size message with full Algorithm B. The left y-axis is number of mods, right y-axis is time in seconds, and x-axis is sample number (32 taken, generating new RSA primitives from random p and q primes of bits from 32 to 1024).

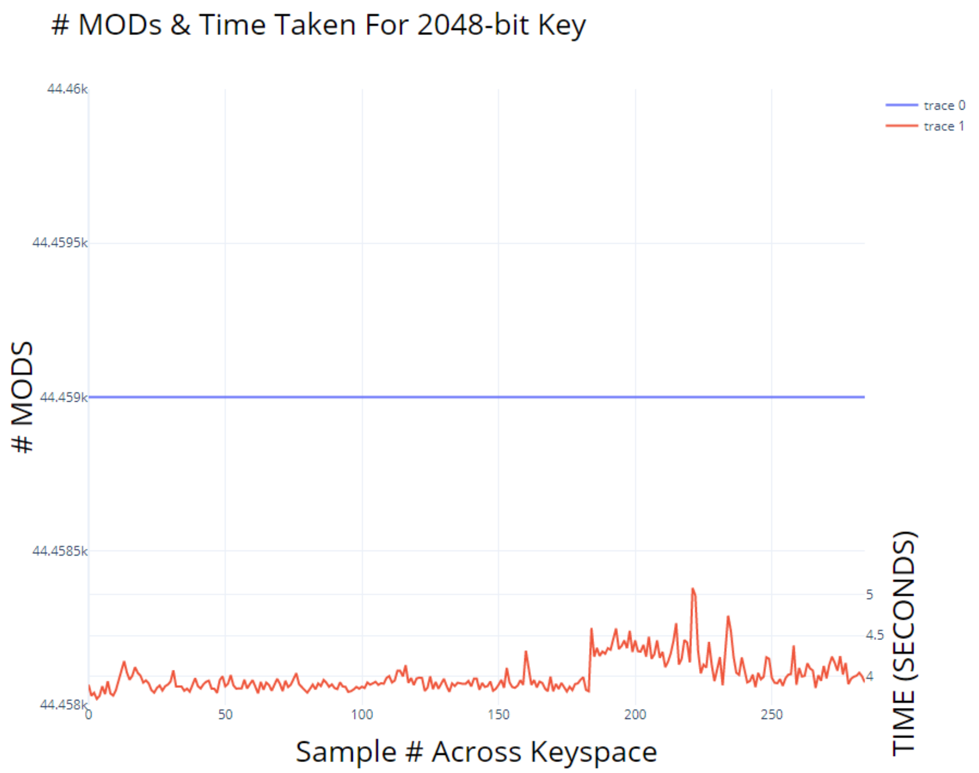

Figure 5: Increasing the message size with a fixed 2048-bit key with full Algorithm B. The left y-axis is number of mods, right y-axis is time in seconds, and x-axis is sample number, up to a message size of 2046 bits.

Considering the sum of these tests, we stand by the initial hypothesis that our Algorithm B would be less vulnerable to the attack outlined for Algorithm A. Given that the basis of the attack on Algorithm A was predicting bits by observing significant variance in timing that correlated with the number of mod operations performed, we believe that our implementation of Granlund's algorithm 6 has successfully silenced the arithmetic operations in the algorithm, and our code for Kocher's blinding has successfully shrouded input messages. Even with the slight variation in real timing (Kocher himself said blinding would follow a relatively normal distribution, but that it hid the input message so that the close observation of message variation could not be used to perform an attack[11]), the constant number of modular operations means that the algorithm did successfully remove execution path variance. By laying Kocher's on top of Granlund's, we see that at the very least blinding does not unravel the gains Granlund's makes in this regard, and so shrouding the input *and* the arithmetic is possible, if desired. In the next section, we will discuss whether we think extra benefit is provided by using both algorithms and whether we think our Algorithm B is still weak to certain side channels.

4 Discussion (W7-W8)

We've shown that combining Granlund's Algorithm 6 and Kocher's blinding, by applying the latter to the inputs to the former, eradicates variability in modulo operations when performing modular exponentiation. Kocher's blinding, additionally, prevents an attacker from being able to make the important distinction between messages as mentioned in outlining Algorithm A. That is, they cannot determine that a message $Y^2 < N$ and $Z^2 < N < Z^3$ for messages $Y, Z \in \mathbb{Z}$ because the messages are modified with the blinding procedure. This means they do not gain important information about which message should have an extra mod operation, and cannot securely make predictions about bits being set based on whether an extra mod operation does or does not occur, because fresh randomization is introduced for each encryption/decryption. However, if the intent of Kocher's blinding scheme is to prevent this step of predicting an extra mod operation, and Granlund's eliminates variation in mod operations, we don't believe Kocher's is necessary on top of Granlund's. In fact, our early stage results testing just Granlund's tended to look almost the same as the results when testing Kocher's on top. Therefore, we don't believe that there is a distinct benefit provided to Granlund's by preceding it with blinding.

The reverse, however, we do not believe to be true. Other types of side-channel attacks, such as attacks targeting the source of randomness on a machine used to generate u, v (`/dev/urandom/` in our case) could result in an implementation of Kocher's blinding that is not careful being undermined. Applying Granlund's under Kocher's blinding acts as an extra layer of defense in case the blinding mechanism specifically were to be targeted. In that manner, we believe combining the two could potentially be beneficial. Ultimately, though, we conclude that in the latter case, it would be simpler to just use Granlund's algorithm, and forgo the use of Kocher's blinding, which is better suited to being applied on top of faster algorithms with strict assurances for randomness and freshness. (In fact, we did some additional testing, which we did not have time to fully graph and analyze, wherein we tested using *only* Kocher's on top of successive squaring. These test programs ran significantly faster than those using Granlund's, and a highly secured implementation of Kocher's can introduce security against mod-counting timing attacks while remaining fast.)

One additional consideration was the trade-offs we made in implementing this algorithm. Our Algorithm B is monumentally slow, sometimes taking 30 seconds for calculations that finish in under a second with successive squaring. Not only this, but we must pre-compute a table of huge powers every time we want to do modular exponentiation, introducing a slew of extra memory consumed by our algorithm. In a real life implementation, we think most users would prefer to use a strong version of Kocher's blinding over top of a more efficient algorithm for modular exponentiation. This saves the trouble of extra memory use and, shown

[11] [3] Kocher, 9.

in some of our experiments, enjoys much faster running times. For scenarios where there is both (a) concern that an implementation of Kocher's may not be robust enough and (b) encryptions are relatively uncommon or the user can wait, we propose that the elimination of conditional execution in Granlund's algorithm 6 makes it a good candidate over Kocher's alone. As mentioned above, we think it is *very* unlikely to need to combine the two.

With the above said, do we believe that our algorithm is still weak to any other kind of side-channel attacks, or that our implementation introduced the possibility of new ones? In fact, Granlund's algorithm has the possibility of introducing side-channel attacks through Euclidean division used to decompose m into base β parts.[12] Granlund discusses being able to do this in a side-channel silent way by using compare-and-subtract, or other faster algorithms which he posits exist. However, these divisions occur in Montgomery's REDC function, which Granlund uses in his implementation of Algorithm 6. In our implementation, we do not use REDC to reduce m into base β. Instead, we used an exorbitantly large β which would cover table entries all the way to the 4096-bit mark. Therefore, our algorithm did not introduce this side-channel.

As for other possible side-channel attacks, we think the elimination of conditional execution of arithmetic protects against even other kinds of side-channel attacks that we were not originally aiming to address. Namely, the low-bandwidth acoustic attack outlined by Genkin, Shamir, and Tromer[13] relies on the conditional execution of Karatsuba's multiplication in GnuPG's implementation of modular exponentiation. Therefore, even audio-based attacks that rely on the information leaked by conditional arithmetic are silenced, because there is no conditional arithmetic in Granlund's algorithm 6.

Ultimately, we have not verified that our cursory, rough Algorithm B implementation is totally resistant to all side-channel attacks (we even noted the possibility of some sort of caching going on due to the curvature of time graphs!), but we think that the elimination of conditional branching, besides for loops which will run a steady amount of iterations, significantly impacts the ability of a variety of side-channel attacks. The discussed side-channel mechanisms tend to rely on the algorithm leaking information by behaving differently for different inputs. Since Granlund's Algorithm 6 does not behave differently for different inputs, and because Kocher's blinding shadows the inputs themselves, applying these methods prevents several classes of side-channel attacks entirely. At the end of the day, we feel that we have learned extensively about the attention to detail and care that needs to be taken to completely eliminate conditional execution, even inadvertent, in order to stop these attacks, and where the particular strengths and weaknesses of efficiency versus determinism become important.

[12][4] Granlund, 7.

[13][6] Genkin, Shamir, Tromer.

5 References

[1] Ben Lynn. "Modular Arithmetic". *Stanford University*, Stanford, California, USA. https://crypto.stanford.edu/pbc/notes/numbertheory/arith.html.

[2] Wing H. Wong. "Timing Attacks on RSA: Revealing Your Secrets through the Fourth Dimension." *San José State University*, San José, California, USA. https://www.cs.sjsu.edu/faculty/stamp/students/article.html.

[3] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." *Cryptography Research, Inc.*, San Francisco, California, USA. https://paulkocher.com/doc/TimingAttacks.pdf.

[4] Torbjörn Granlund. "Defeating modexp side-channel attacks with data-independent execution traces." *Royal Institute of Technology: Theoretical Computer Science Group*, Stockholm, Sweden. https://gmplib.org/ tege/modexp-silent.pdf.

[5] Techie Delight. "Find maximum number without using conditional statement or ternary operator." *Techie Delight.* https://www.techiedelight.com/find-maximum-number-without-using-conditional-statement-ternary-operator/

[6] Daniel Genkin, Adi Shamir, Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis." *Tel Aviv University.* http://www.cs.tau.ac.il/ tromer/papers/acoustic-20131218.pdf.