Time as a Key: Breaking Rhysida Ransomware with the Attacker's Own Ciphertext

Adam Taguirov
Independent Researcher
adam.taguirov@sigreturn.com

***Abstract*—Rhysida is a ransomware-as-a-service operation active since May 2023 that has claimed more than 250 victims across education, healthcare and professional services. Its Windows encryptor seeds the C library pseudo-random number generator with `srand(time(0))`, and every per-file secret produced during a run derives from that single value. The encryption timestamp, a 32-bit counter of seconds, is the only unknown input, and recovering it regenerates the AES-256 key and initialisation vector of every file the run touched. The encryptor wraps each per-file key and IV under a hardcoded RSA public key using OAEP padding, and it draws the OAEP seed from the same generator, so the RSA blob appended to every encrypted file is reproducible byte for byte by anyone holding the sample. We use that blob as a validation oracle: a candidate key is confirmed by re-running the encryptor's own RSA step and comparing the result against the stored bytes. The test needs no plaintext and no file-format heuristic, and it never writes a wrong decryption over a victim's file. Implementing the recovery from the paper alone reproduces both ciphertexts of a real encrypted file's trailer byte for byte, and recovers the seed at one RSA operation per candidate second when the victim's file set is available. Seventeen variants carry the flaw. A later C/C++ build resists the recovery, and the group's parallel PowerShell encryptor, which draws randomness from `RNGCryptoServiceProvider`, was never affected. The analysis and the decryptor date from 2023.**

***Index Terms*—ransomware, cryptanalysis, pseudo-random number generation, reverse engineering, malware analysis, digital forensics, incident response**

I. Introduction

Rhysida surfaced in May 2023 and has been tracked since November 2023 in a joint advisory from the FBI, CISA and MS-ISAC [1]. Leak-site tracking puts it at 273 claimed victims across 39 countries, led by education, healthcare and professional services [2]. The best-documented of them is the British Library, which published its own incident review [3]. Like almost every ransomware family in production use it encrypts with a hybrid scheme: a fresh symmetric key per file, the file encrypted under it, and the key wrapped with an RSA public key compiled into the binary. Done properly, this leaves a victim two options, restore from backup or pay.

Rhysida does not do it properly. Listing 1 shows the entry point of the vulnerability.

```
.text:00000000004197C0 mov  ecx, 0    ; Time
.text:00000000004197C5 call time
.text:00000000004197CA mov  ecx, eax  ; Seed
.text:00000000004197CC call srand
```

Listing 1. The seeding of the encryptor's PRNG, at the top of `main` in sample S0.

The C library generator is seeded with the current time in seconds. Every value the encryptor treats as random from that point on, including the per-file AES-256 keys and initialisation vectors, is a deterministic function of one 32-bit integer that can be read off the victim's file system.

That much has since been reported publicly [4]. Having generated a key and an IV from the broken generator, the encryptor wraps them under the hardcoded RSA public key with OAEP padding, and it passes the same generator state to the padding routine as the source of the OAEP seed. OAEP is randomised by design. The RSA blob the encryptor appends to every victim file is therefore reproducible, byte for byte, by anyone holding a copy of the sample and a guess at the timestamp.

This gives an exact validation oracle. A candidate key is confirmed by re-running the encryptor's own RSA step over it and comparing the result against the bytes already sitting in the encrypted file. The comparison is over the full RSA modulus length, and a false positive is not practically possible.

A. Contributions

- A static analysis of the Rhysida encryptor, taken from a sample the operator shipped with debugging symbols: the key schedule, the thread pool, the file traversal, the exclusion sets, the intermittent encryption layout and the trailer format (Section IV).
- A statement of the determinism chain: which values are a function of the seed, how many `rand()` draws each stage consumes, and the one thing it does not determine, the assignment of files to threads (Section V).
- Ciphertext-verified key recovery. We give the recovery algorithm, its cost model and its inputs (Section VI).
- An end-to-end verification: the recipe reimplemented from the paper alone reproduces both 512-byte ciphertexts of a real encrypted file's trailer, with measured search costs and a worked example (Section VII).
- Confirmation of the flaw across seventeen variants, and an account of the two builds the recovery does not reach (Section VII).
- A comparison of method with the prior public analysis, on how a candidate key is confirmed (Section VIII).

B. Provenance

The analysis in this paper was carried out in May 2023, weeks after Rhysida first surfaced, and a working decryptor

followed. None of it could be published at the time.

The public research that later emerged, and the decryptors built on it, arrived independently and confirmed the same underlying flaw.

II. Background

A. Rhysida

Rhysida operates as ransomware-as-a-service under a double-extortion model with a Tor leak site. The earliest sample we observed is:

```
SHA-256: a864282fea5a536510ae86c77ce46f7827
         687783628e4f2ceb5bf2c41b8cd3c6
```

We number it S0, a 64-bit PE.

B. The primitives Rhysida uses

Every cryptographic operation in the encryptor comes from a single external library, libtomcrypt [5].

1) ChaCha20 as a PRNG: libtomcrypt exposes ChaCha20 [6] as a PRNG descriptor, whose `start`, `ready`, `add_entropy` and `read` entry points the encryptor calls.

2) AES-256 in CTR mode: File data is encrypted with AES [7] in counter mode [8], through libtomcrypt's `ctr_start`, `ctr_setiv` and `ctr_encrypt`. CTR turns the block cipher into a stream cipher. libtomcrypt's `ctr_decrypt` is a one-line function whose body is a call to `ctr_encrypt`.

3) RSA with OAEP: Per-file secrets are wrapped with `rsa_encrypt_key_ex` using PKCS#1 v2.2 OAEP padding [9]. OAEP is a randomised padding scheme: it draws a seed of one hash-length from a PRNG, expands that seed through the mask generation function MGF1 into a mask as long as the message, and XORs the two together, so two encryptions of the same message under the same public key give different ciphertexts. libtomcrypt requires the caller to hand it a `prng_state` and a PRNG index.

4) CHC as the OAEP hash: The encryptor registers libtomcrypt's CHC construction, which manufactures a hash out of a block cipher, and binds it to AES. Its digest length is the cipher's block length, so 16 bytes rather than SHA-256's 32.

C. `srand` and `rand`

The C standard defines `rand` as a pseudo-random sequence generator whose state is set by `srand`, and guarantees that for a fixed implementation, the same seed produces the same sequence [10].

The concrete sequence is implementation-defined. Every sample in the corpus imports `msvcrt.dll` and takes `rand`, `srand`, `_getcwd` and `_beginthreadex` from it, which identifies a MinGW-w64 build running against the Microsoft C runtime. Its `rand` is the linear congruential generator

$$\begin{aligned} x_{j+1} &= (214013\, x_j + 2531011) \bmod 2^{32}, \\ \mathrm{out}_j &= (x_{j+1} \gg 16) \wedge \texttt{0x7fff} \end{aligned} \tag{1}$$

returning the middle 15 bits of its state. Replaying the same seed through glibc's `rand`, which uses an additive-feedback generator over a 31-element state array, produces a different sequence and recovers nothing.

TABLE I
THE CORPUS. "KEY" GROUPS SAMPLES THAT CARRY THE SAME EMBEDDED RSA PUBLIC KEY. "PUBLIC" MARKS THE FIFTEEN WHOSE HASHES ARE ALREADY DOCUMENTED IN OPEN SOURCES. THOSE HASHES ARE IN APPENDIX C.

ID	SHA-256 (prefix)	Key	Public
S0	`a864282fea5a5365…`	K1	yes
S1	`d5c2f87033a5baee…`	K2	yes
S2	*withheld*	K3	–
S3	`3518195c256aa940…`	K4	yes
S4	`67a78b39e760e346…`	K1	yes
S5	`250e81eeb4df4649…`	K1	yes
S6	`a78fa1ecadc46870…`	K5	yes
S7	`258ddd78655ac058…`	K6	yes
S8	`3d2013c2ba0aa1c0…`	K7	yes
S9	*withheld*	K2	–
S10	`2c5d3fea7ad3c9c4…`	K8	yes
S11	`f6f74e05e24dd2e4…`	K9	yes
S12	`0bb0e1fcff8ccf54…`	K7	yes
S13	`8061bb999a0f5d31…`	K10	yes
S14	`1a9c27e5be8c58da…`	K11	yes
S15	`b55ecbddcbed9164…`	K12	yes
S16	`0050a69d6e93eddc…`	K1	yes

III. Analysis Setup and Corpus

The question this work set out to answer was whether the Rhysida encryptor contained a cryptographic or logic flaw permitting partial or total recovery of encrypted files.

Static analysis was done in IDA Pro 7.5.200728, and the decryptor was written in C and built for Windows x64 and Linux x64. The measurements in Section VII were taken later, under Debian 13 with GCC 14.2.

A. Corpus

Strains are numbered by order of first appearance, whether through a victim or through open sources. Table I gives all seventeen, sixteen of them 64-bit Windows PE builds and one, S8, a 32-bit i386 build. All seventeen carry the flaw.

Later strains have not been examined, so seventeen is a lower bound.

B. The reference sample

The analysis below rests on S0. This version of Rhysida ships with debugging symbols, which makes it considerably easier to analyse and gives us the function names used throughout this paper. It is also the sample the decryptor was built against, and it was then adapted to the other versions as they appeared. Most later builds are stripped, but the structure is the same and the analysis carries over.

One variant is not in Table I: the PowerShell strain, which Section VII-F covers.

IV. Inside the Rhysida Encryptor

Execution drops into the encryptor through `main`, which sets everything up before any of the victim's files get touched. Listing 2 is that setup. Two things jump out of it.

```
v3 = time(0i64);
srand(v3);
getcwd(cwd, 260);
GetSystemInfo(&sysinfo);
PROCS = sysinfo.dwNumberOfProcessors;
printf("Number of procs %ld\n", sysinfo.dwNumberOfProcessors);
prngs                 = (prng_state *)malloc(17648i64 * PROCS);
PRNG_IDXS             = (int *)malloc(4i64 * PROCS);
QUERY_FILE_THREAD_IDS = (pthread_t *)malloc(8i64 * PROCS);
thread_is             = (int *)malloc(4i64 * PROCS);
QUERY_FILE_POSS       = (int *)malloc(4i64 * PROCS);
QUERY_FILES           = (char ***)malloc(8i64 * PROCS);
QUERY_FILE_LOCKEDS    = (int *)malloc(4i64 * PROCS);
MUTEXES               = (pthread_mutex_t *)malloc(8i64 * PROCS);
pthread_mutex_init(&MUTEX_PRNG, 0i64);
for ( thread_i = 0; thread_i < PROCS; ++thread_i )
{
  pthread_mutex_init(&MUTEXES[thread_i], 0i64);
  QUERY_FILE_POSS[thread_i] = -1;
  v4 = &QUERY_FILES[thread_i];
  *v4 = (char **)malloc(0x2000ui64);
  for ( files_i = 0; files_i <= 1023; ++files_i )
  {
    v5 = &QUERY_FILES[thread_i][files_i];
    *v5 = (char *)malloc(0x1000ui64);
  }
  QUERY_FILE_LOCKEDS[thread_i] = 0;
  thread_is[thread_i] = thread_i;
}
options = (Options *)malloc(0x18i64);
parseOptions(argc, (char **)argv, options);
```

Listing 2. The head of `main` in S0, as recovered by the decompiler.

A. Entry point and initialisation

`srand` is seeded with `time(0)`. The PRNG is initialised straight from the current timestamp, in seconds since 1 January 1970 at 00:00.

`GetSystemInfo` is called, then `sysinfo.dwNumberOfProcessors` is read. Rhysida grabs the number of logical processors on the victim's machine and puts it in a global, `PROCS`, which it uses later to parallelise the encryption. We write P for that count throughout. Every array the encryptor allocates is sized against it, including `prngs`, the array of libtomcrypt PRNG states, at `0x44F0` = 17648 bytes each.

Next we see a loop that runs a counter up to the number of logical processors found earlier, initialising the per-thread state for each. A second loop further on spawns one thread per logical processor, each entering `processFiles`, and each is later used to generate the file encryption keys asynchronously. This is exactly why the victim machine's logical processor count matters when the keys are regenerated, and that number is usually standard anyway (4, 8, 16, 32, 64…). Each thread also gets a work queue of `0x2000/8` = 1024 path slots and its own mutex in `MUTEXES`.

B. Option parsing

Once that initialisation loop is done, the program calls an internal function named `parseOptions`, which parses the arguments the attacker passed to the program, used to toggle certain internal options of the encryptor or switch its operating mode. The full decompilation is in Appendix B. Two parameters stand out.

`-d` lets the attacker point the program at a specific directory. The path is stored in a `directory_modifier` variable, whose value then gets written into the program's internal options.

`-sr` tells the program to delete itself once it is done running. The boolean is held in a `self_remove_modifier` variable and likewise written into the internal options.

There is a bug in S0's `parseOptions`. The function sets `options->is_self_remove = 1` unconditionally at the top, and the `-sr` branch only sets it to 1 again. In this build, self-deletion is therefore the default behaviour. The self-deletion command string is present in only seven of the seventeen samples in Table I.

C. Cryptographic initialisation

Further along the execution flow we hit a series of routines that initialise the encryptor's cryptographic parameters.

`init_prng` initialises the pseudo-random number generator, and Section IV-F covers it.

`rsa_import` imports, among other things, a public RSA key referenced earlier in the code, along with its size. The key is a DER blob written in hard in the encryptor, at the symbols `__PUB_DER` and `__PUB_DER_LEN`.

`register_cipher` then `find_cipher` are called in succession to set up the AES cipher. The descriptor registered is `_refptr_aes_enc_desc`, the string looked up is `"aes"`, and the resulting index is stored in the global `CIPHER`.

`register_hash`, `chc_register` then `find_hash` are called in succession to set up the hash function used. `chc_register` is passed the `CIPHER` index recovered a moment earlier and the string looked up is `"chc_hash"`, so the hash is the CHC construction over AES. Its index lands in `HASH_IDX`, which is passed to `rsa_encrypt_key_ex` later, and that makes CHC-AES the hash inside OAEP and MGF1.

```
int __cdecl addFileToQueue(char *filename, int thread_n)
{
  int result;

  if ( pthread_mutex_trylock(&MUTEXES[thread_n]) )
    return 0;
  if ( QUERY_FILE_LOCKEDS[thread_n]
    || QUERY_FILE_POSS[thread_n] == 1023 )
  {
    pthread_mutex_unlock(&MUTEXES[thread_n]);
    result = 0;
  }
  else
  {
    QUERY_FILE_LOCKEDS[thread_n] = 1;
    strcpy(QUERY_FILES[thread_n][++QUERY_FILE_POSS[
  thread_n]],
           filename);
    QUERY_FILE_LOCKEDS[thread_n] = 0;
    pthread_mutex_unlock(&MUTEXES[thread_n]);
    result = 1;
  }
  return result;
}
```

Listing 4. `addFileToQueue`.

D. Walking the file system

The program then moves on to walking the system's files, through a parent function `openDirectoryNR` that takes the path of the directory to recurse into:

```
void __cdecl openDirectoryNR(char *directory_name);
```

1) How directories are selected: The argument, the full path to the directory to walk and encrypt, depends on the `-d` option the attacker passes to the program. For example, `-d C:\Users\test\Downloads` tells the program to encrypt only the `Downloads` folder of the user `test`. If no `-d` parameter is given, meaning the default behaviour, the program iterates over every letter from `A` to `Z` and tries to recursively encrypt every drive mounted on the system, skipping the letters where no mount point exists. So the main system drive, often `C:`, gets encrypted, along with any other data disks and mounted external storage. Listing 3 is what that looks like.

```
if ( *options->directory )
{
  openDirectoryNR(options->directory);
}
else
{
  drive = (char *)malloc(0x1000ui64);
  for ( drive_letter = 'A'; drive_letter <= 'Z'; ++
  drive_letter )
  {
    sprintf(drive, "%c:/", (unsigned int)drive_letter);
    openDirectoryNR(drive);
  }
  free(drive);
}
```

Listing 3. Drive selection.

For simplicity we will not break down `openDirectoryNR` in detail. It is a file-traversal function: it works through a queue holding the folders to visit one after another, while regular files are pulled out of the traversal and added to the per-thread array `QUERY_FILES` by the `addFileToQueue` function, at the index held in `QUERY_FILE_POSS`.

Listing 4 is `addFileToQueue`.

`pthread_mutex_trylock` asks for the queue's lock and returns immediately either way, zero if it got it and non-zero if another thread was holding it, and Rhysida treats non-zero as "skip this thread". The offer also fails if the queue is marked locked or full at 1024 entries. Whether it succeeds depends on what the other threads were doing at that moment and on nothing else.

2) Excluded directories: Some folders are skipped by the encryptor, via an array of directory paths named `exclude_directories` that the `isDirectoryExcluded` function checks against. These are mostly system and boot directories: leaving them untouched keeps the machine bootable and usable enough for the victim to actually read the ransom note and pay. The array is stored as integers in the binary. Once exported and converted back to strings we get eleven paths, listed in Appendix A.

E. Encrypting files

Across multiple threads, one per logical processor on the system, the `processFiles` function is called to handle the files assigned to each thread. It walks the array of files to encrypt, extracts the file path's name, checks whether the file is actually a legitimate target with `isFileExcluded`, and encrypts it where appropriate with `processFileEnc`:

```
if ( !isFileExcluded(file_name) )
{
  ++global_statistics.all_count;
  processFileEnc(file_name, 0, *thread_n);
}
```

The third argument is the thread index.

1) Excluded files: Some files are left out of encryption too. In `isFileExcluded` we see filtering on file extensions, driven by an integer array `exclude_extensions` that holds the extensions to skip. As with the excluded directories, these are mostly executables and system files: encrypting them would risk breaking the OS and leaving the machine unbootable. The converted list has 27 entries and is in Appendix A.

F. A quirk in the random number generation

Before getting into the encryption process itself, it is essential to understand how Rhysida generates randomness.

`init_prng` is called once for a standalone generator, and then once more per thread that can run simultaneously during execution:

```
init_prng(&prng, &PRNG_IDX);
for ( thread_i = 0; thread_i < PROCS; ++thread_i )
{
  if ( init_prng(&prngs[thread_i], &PRNG_IDXS[thread_i])
   )
    goto LABEL_8;
}
```

Each thread maps to a logical processor core. The count of `init_prng` calls is $P + 1$, not P, and the first generator is never used to produce a file key.

Listing 5 is the function. It makes several calls into the external libtomcrypt library, notably the ChaCha20 random-string generation functions, but also calls to `rand`, which acts directly on the value passed earlier to its seeder, `srand`. In

```
int __cdecl init_prng(prng_state *prng_val, int *n)
{
  int v3;                           // eax
  unsigned __int8 prng_entr[40];    // [rsp+20h] [rbp-50h] BYREF
  unsigned int read_len;            // [rsp+54h] [rbp-1Ch]
  unsigned __int8 *buf;             // [rsp+58h] [rbp-18h]
  int buf_len;                      // [rsp+64h] [rbp-Ch]
  int err;                          // [rsp+68h] [rbp-8h]
  int i;                            // [rsp+6Ch] [rbp-4h]

  *n = register_prng(refptr_chacha20_prng_desc);
  if ( *n == -1 )
    return 1;
  if ( (unsigned int)chacha20_prng_start(prng_val) )
    return 2;
  err = chacha20_prng_ready(prng_val);
  if ( err )
    return 3;
  for ( i = 0; i <= 39; ++i )
    prng_entr[i] = rand() * (*(_BYTE *)n + i + 1);
  err = chacha20_prng_add_entropy(prng_entr, 40i64, prng_val);
  if ( err )
    return 4;
  v3 = rand();
  buf_len = (unsigned __int8)(((unsigned int)(v3 >> 31) >> 24) + v3)
          - ((unsigned int)(v3 >> 31) >> 24) + 1;
  buf = (unsigned __int8 *)malloc(buf_len);
  read_len = chacha20_prng_read(buf, 8i64, prng_val);
  free(buf);
  return 0;
}
```

Listing 5. `init_prng`. Every call consumes exactly 41 `rand()` draws: 40 in the entropy loop and one after it.

our case, that seed is the timestamp passed to `srand` at the start of the program.

The 40 entropy bytes are `(uint8_t)(rand() * (idx + i + 1))`, so the only thing that varies between threads is which 40 `rand()` outputs land in the loop, since `idx`, the libtomcrypt registry slot returned by `register_prng`, is the same on every call. `chacha20_prng_ready` is called *before* `chacha20_prng_add_entropy`, which is unusual and changes the resulting state. The trailing `rand()` sizes a `malloc` whose contents are never used, but it still advances the generator, so each call consumes exactly 41 draws. And the final `chacha20_prng_read` discards 8 bytes, so every thread's keystream starts 8 bytes in.

The global array fed by this function, named `prngs` and sized to the number of processors, holds the various initialisation values for the ChaCha20 random-string generator. It is used throughout the rest of the program, in particular to generate the encryption keys.

The key thing to remember here: the processor count decides how the `rand` sequence gets carved up. With P threads the encryptor draws $41(P+1)$ values and hands draws $[41(i+1),\ 41(i+2))$ to thread i.

G. The encryption routine

Rhysida's encryption algorithm follows a specific process that we find in every strain we analysed. A victim file is encrypted in the `processFileEnc` function, called on each targeted file in turn, taking the file path and name as its argument. We will not detail every cryptographic function involved. To keep things simple, we focus only on what is essential to explain how the vulnerability is exploited and how the decryptor was built.

In short: the key and IV for each file are produced by a ChaCha20 string-generation function. Those values are encrypted with RSA, whose private decryption key is held solely by the attacker. The file is then encrypted with the generated key and IV using the CTR algorithm, and these RSA-encrypted values are stored at the end of the encrypted victim file. The ChaCha20, RSA and CTR algorithms are all implementations from a single external library, libtomcrypt.

1) The encryption process, step by step: Listing 6 is the path Rhysida takes.

a) Generate a 32-byte key and a 16-byte initialisation vector with `chacha20_prng_read`, then initialise the cipher with that key and IV using `ctr_start`: The disassembly in Listing 7 gives the parameters:

b) Set the IV for the CTR algorithm with `ctr_setiv`: Sixteen bytes, into the same `symmetric_CTR` context.

c) Encrypt the previously generated key and IV using RSA and a public key hardcoded into the executable: `rsa_encrypt_key_ex` handles this encryption of the secrets, and the resulting encrypted values are written to the end of the victim's file, after an `fseeko64(Stream, 0, SEEK_END)`. The padding argument is `2`, which is libtomcrypt's `LTC_PKCS_1_OAEP`, and the OAEP label is the 11-byte string `"Rhysida-0.1"`.

The embedded key is RSA-4096, so each ciphertext is 512 bytes. Reading the trailer of an encrypted file byte by byte gives the layout in Table II: the two ciphertexts, each followed by a four-byte little-endian length field holding `0x200`, then a final word equal to 1. The total is `0x40C` = 1036 bytes. The two length fields and the trailing word hold the same values in all ten files of the corpus used in Section VII-B.

d) Encrypt the file block by block: Each block goes through a call to `ctr_encrypt` using the generated key and

```
1  /* 1. key and IV, from this thread's ChaCha20 state */
2  chacha20_prng_read(key, 32i64, prngs + 17648 * a3);
3  chacha20_prng_read(iv,  16i64, prngs + 17648 * a3);
4  ret = ctr_start(CIPHER, iv, key, 32, 14, 0, init_ctr);
5  ret = ctr_setiv(iv, 16, init_ctr);
6
7  /* 2. wrap the key under the embedded RSA public key */
8  HIDWORD(Size) = 32;  HIDWORD(ElementSize) = 4096;
9  ret = rsa_encrypt_key_ex(key, 0x20ui64, encrypted_key,
10                           &ElementSize, "Rhysida-0.1", 0xB,
11                           prngs + 17648 * a3,
12                           PRNG_IDX, HASH_IDX, 2, &rsa_public_key);
13 ret_2 = 1 - fwrite(encrypted_key, HIDWORD(ElementSize), 1ui64, Stream);
14
15 /* 3. same for the IV */
16 LODWORD(Size) = 16;  LODWORD(ElementSize) = 4096;
17 ret = rsa_encrypt_key_ex(iv, 0x10ui64, encrypted_iv,
18                           &ElementSize, "Rhysida-0.1", 11,
19                           prngs + 17648 * a3,
20                           PRNG_IDX, HASH_IDX, 2, &rsa_public_key);
21 ret_2 = 1 - fwrite(encrypted_iv, ElementSize, 1ui64, Stream);
```

Listing 6. The per-file encryption core in `processFileEnc`, reassembled from the decompilation. `a3` is the thread index and `17648` is `sizeof(prng_state)`. The generator on lines 2, 3, 11 and 19 is the same state.

```
mov  edx, 32          ; 32-byte key
call chacha20_prng_read
mov  edx, 16          ; 16-byte IV
call chacha20_prng_read
lea  rax, CIPHER
mov  eax, [rax]
lea  r8,  [rbp+cipher_key]
lea  rdx, [rbp+cipher_IV]
lea  rcx, [rbp+ctr]
mov  [rsp+var_1032E0], rcx
mov  [rsp+var_1032E8], 0
mov  dword ptr [rsp+var_1032F0], 0Eh
mov  r9d, 20h
mov  ecx, eax
call ctr_start
```

Listing 7. `ctr_start` operands. `r9d = 20h` is a 32-byte key and the stacked `0Eh` is 14 rounds, so the cipher is AES-256.

TABLE II
THE 1036-BYTE TRAILER APPENDED TO EVERY ENCRYPTED FILE, OFFSETS GIVEN FROM THE END OF THE FILE.

Offset	Size	Content
−1036	512	RSA-OAEP wrap of the 32-byte AES key
−524	4	`0x00000200`, length of the field above
−520	512	RSA-OAEP wrap of the 16-byte IV
−8	4	`0x00000200`, length of the field above
−4	4	`0x00000001`

IV, which differ for every file. Listing 8 is the loop.

2) Intermittent encryption: Rhysida does not encrypt the whole file if it is larger than 1 048 576 bytes (`0x100000` in hex), which is the block size the attacker uses. The constant is assigned directly:

```
mov [rbp+file_process_block], eax
mov [rbp+file_process_block_size], 100000h
```

When a file is larger than the block size, the encryptor tries to fit in as many blocks as possible, up to a limit of 4. So if a file is bigger than one block but too small to hold two, Rhysida only encrypts the portion matching the first block and leaves the rest in clear. In the other case, if the file is large enough to comfortably hold all 4 blocks, Rhysida encrypts exactly the space taken by those 4 blocks and skips the gaps between them, leaving those untouched. The 4 blocks are spread across the entire length of the file, each separated by an equal-sized region that stays unencrypted.

Write $B = \texttt{0x100000}$ for the block size. A payload of $s \leq B$ bytes is encrypted whole. Above that the encryptor writes $m = \min(4, \lfloor s/B \rfloor)$ blocks at stride $v = \lfloor s/m \rfloor$, which is the `v19` of Listing 8. Block j covers $[jv,\ jv + B)$. Since $mv \leq s$, a clear region of $v - B$ bytes follows every block, the last one included, so the tail of a large file is never encrypted. Figure 1 shows the five cases. The technique spread through ransomware development from 2021 onward, traded against detectability rather than adopted for any cryptographic reason [11].

All the blocks are encrypted under the same CTR context in a single pass, with the counter running across them and never reset between them. The gaps consume no keystream, so block j starts at byte jv of the file but at byte jB of the keystream. A decryptor that seeks in the keystream as if the gaps had been encrypted produces garbage from the second block on.

The program then renames the file by appending the `.rhysida` extension:

```
rename(filename_extension_rhysida, filename_original);
```

H. Ransom note and end of execution

Once file encryption is done, the program deletes itself. The mechanism is a detached PowerShell one-liner:

```
for ( i = 0; i < max_blocks; ++i )
{
  sub_448D50(Stream, v19 * i, 0);
  v10[0] = fread(*&v10[1], 1ui64, block_size, Stream);
  if ( v13 != block_size && !v10[0] )
  {
    ret_2 = 1;
    break;
  }
  ret = ctr_encrypt(*&v10[1], *&v10[1], v10[0], init_ctr);
  if ( ret )
  {
    ret_2 = 1;
    i = 4;
  }
  else
  {
    sub_448D50(Stream, v19 * i, 0);
    fwrite(*&v10[1], v10[0], 1ui64, Stream);
  }
  (sub_41F130)(*&v10[1], 8i64);
}
```

Listing 8. The block loop. `v19` is the stride between blocks, and `max_blocks` never exceeds 4.

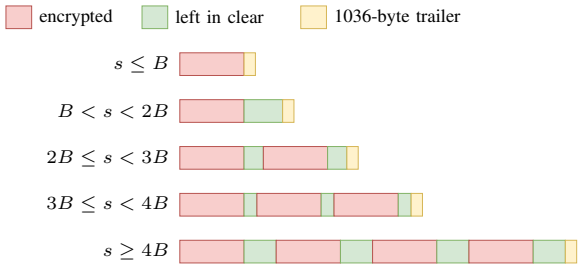

Fig. 1. Intermittent encryption by payload size s, with block size $B = \texttt{0x100000} = 1\,048\,576$ bytes.

```
strcpy(command,
  "cmd.exe /c start powershell.exe -WindowStyle Hidden "
  "-Command Sleep -Milliseconds 500; Remove-Item -Force
  -Path \"");
strcat(command, cwd);
*(_WORD *)&command[strlen(command)] = '\\';
strcat(command, options->program);
strcat(command, "\" -ErrorAction SilentlyContinue;");
...
system(command);
```

Where the routine is present the encryptor binary is normally gone from the machine it encrypted, and the sample has to come from elsewhere in the intrusion or from open sources.

As encryption progresses, ransom notes in PDF format are dropped into each encrypted directory. When encryption finishes, the program updates the Windows wallpaper, replacing it with a ransom note as well, via the `setBG` function. Since this process is not essential to the analysis here, we will not go into the details.

V. The Vulnerability

The C version of Rhysida has a vulnerability that lets us decrypt the victim's files with no prior knowledge of the key, and with no special requirement beyond access to the victim's machine or the encrypted files.

A. Explaining the vulnerability

Every bit of randomness in the encryptor comes from a single source, initialised by `srand` with a seed, which here is the timestamp. Using a timestamp as the seed for random generation in a cryptographic context completely undermines the crypto chain that follows and makes the encryption void and reversible.

The reason is that the keys and IVs the program generates for each file all derive from `rand`, which itself relies on the value handed to `srand`. If you call `rand` several times in a row while always passing the same value to `srand`, you will always get the same sequence back:

```
srand(5);
rand(); // generated value: 42
rand(); // generated value: 77
rand(); // generated value: 92
rand(); // generated value: 8
...
```

Run this code several times in a row and the `rand` calls produce the same values every time. Knowing the value passed to `srand`, the seed, lets you predict every random value the program generates, including the encryption keys and IVs. And all we need to recover that value is the timestamp, even an approximate one, of when the victim's files were encrypted.

B. What the seed determines

Figure 2 gives the chain.

Let t be the value returned by `time(0)` at process start and P the logical processor count. Write $r_0, r_1, \ldots$ for the sequence produced by `rand` after `srand(`t`)`. In S0, `init_prng` is the only function in the encryptor that calls `rand` at all.

Proposition 1. *The ChaCha20 state of thread i after initialisation is a function of t and i alone. It does not depend on P.*

Each `init_prng` call consumes exactly 41 draws: 40 in the entropy loop and one for the discarded `malloc` size. The calls are made sequentially from `main`, before any worker thread is created, and the standalone generator is created first, so thread i is initialised by the call after the i preceding thread generators and consumes draws $[41(i{+}1),\ 41(i{+}2))$ regardless of how many calls follow it. P decides how many states exist, not what any one of them contains.

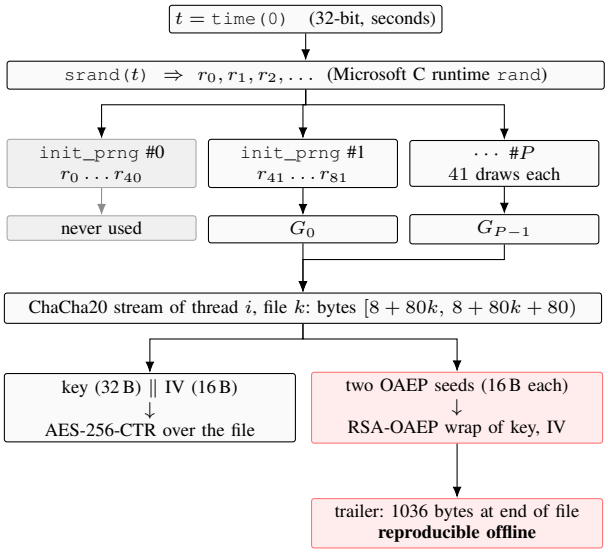

Fig. 2. The determinism chain. On the shaded path, the OAEP randomness comes out of the same stream as the key it is padding.

In consequence we do not need the victim's exact processor count, we need an upper bound on it. Generating states for $i < 64$ covers every machine with at most 64 logical processors, because the surplus states never match.

Proposition 2. *Given t, every byte of key material the run produces can be computed offline, including the OAEP padding randomness.*

The samples link against a libtomcrypt build that provides the ChaCha20 PRNG descriptor, `chacha20_prng_desc`, since `init_prng` calls its `start`, `ready`, `add_entropy` and `read` entry points by name.

`chacha20_prng_start` zeroes a 40-byte entropy buffer and clears the `ready` flag. `chacha20_prng_ready` is then called while that buffer is still all zeros, so it keys ChaCha20 with a 32-byte zero key and an 8-byte zero nonce and sets `ready`. Only then does `init_prng` add its entropy, and because `ready` is now set, `chacha20_prng_add_entropy` takes the rekey path: it pulls 40 bytes of keystream from that zero-keyed state, XORs the caller's 40 bytes into them, and uses the result as the new key and nonce.

The 40 bytes it pulls are therefore a constant, the head of the ChaCha20 keystream under an all-zero key:

```
KS0 = 76b8e0ada0f13d90405d6ae55386bd28bdd219b8
      a08ded1aa836efcc8b770dc7da41597c5157488d
```

Each thread's ChaCha20 key and nonce are then that constant XORed with the 40 entropy bytes that thread drew. Written out, the state of thread i is

$$\begin{aligned} G_i &= \mathrm{KS0} \oplus E_i, \\ E_i[j] &= \big(r_{41(i+1)+j} \cdot (idx + j + 1)\big) \bmod 256 \end{aligned} \tag{2}$$

for $j \in [0, 40)$, with $G_i[0..32)$ the ChaCha20 key, $G_i[32..40)$ the 64-bit nonce, counter zero and 20 rounds. Here idx is the libtomcrypt PRNG registry index, which is the same on every call because re-registering a descriptor returns its existing slot. Its value is 0: ChaCha20 is the only PRNG descriptor the encryptor ever registers.

C. The per-file stream layout

`init_prng` ends with `chacha20_prng_read(buf, 8, prng_val)`, whose output is freed unread. libtomcrypt buffers leftover keystream inside the state, so successive reads are contiguous in one stream: the discard moves each thread's stream forward by 8 bytes.

Each file then consumes exactly 80 bytes of its thread's stream, in this order:

Offset	Length	Use
0	32	AES-256 file key
32	16	CTR initialisation vector
48	16	OAEP seed, wrapping the key
64	16	OAEP seed, wrapping the IV

The first two come from the explicit `chacha20_prng_read` calls in Listing 6. The last two are read *inside* `rsa_encrypt_key_ex`: libtomcrypt's OAEP encoder draws a seed of one hash length from the PRNG the caller handed it, and the caller here handed it `prngs + 17648 * a3`, the same state that produced the key. The hash is CHC bound to AES, whose digest size `chc_register` sets to the cipher's block length, so one hash length is 16 bytes.

So for thread i, the k-th file it processes uses stream bytes $[8 + 80k,\ 8 + 80(k + 1))$.

D. The consequence: the padding is not random either

Rhysida hands the OAEP encoder the same broken generator it used to produce the message. The seed is not random, it is stream byte $8 + 80k + 48$ of a stream that t determines. Every input to the encoding is therefore known to anyone holding the sample and a guess at t: the message (the file key), the seed, the label (`"Rhysida-0.1"`), the hash (CHC-AES) and the public key (the DER blob at `__PUB_DER`). OAEP encoding is deterministic given those, and RSA encryption with a public exponent is deterministic given the encoded block.

The bytes Rhysida appends to the victim's file are therefore reproducible offline, exactly, without the private key.

E. What the seed does not determine

Which thread encrypts which file is decided at traversal time by `addFileToQueue` (Listing 4), and so is its position in that thread's work.

The pair (i, k) is therefore the unknown, and the recovery has to search over it. Everything else follows from t.

VI. Recovering the Keys

A. The validation oracle

Three things come out of the sample and never change: the DER public key at `__PUB_DER`, giving the modulus n and the public exponent e, which is RSA-4096 in all twelve distinct keys the corpus carries; the OAEP label L = `"Rhysida-0.1"`; and the OAEP hash H, CHC bound to AES, whose digest length is 16 bytes.

Write $\mathrm{Gen}(t, i, k)$ for the four values a run seeded with t hands to thread i for its k-th file: the AES key, the IV, and the two OAEP seeds, read off as the 80 bytes at offset $8 + 80k$ of that thread's stream. Write $\mathrm{Wrap}(\mu, \sigma)$ for the encryptor's own wrapping step: OAEP-encode μ under that label and hash using σ as the seed, then raise it to the public exponent.

Every encrypted file ends with the 1036-byte trailer of Table II, whose first 512 bytes are the wrapped key and whose bytes 516 to 1027 are the wrapped IV.

Proposition 3 (Oracle)**.** Wrap *applied to a candidate key and its OAEP seed equals the wrapped key stored in a file if and only if that file was encrypted by a run seeded with t, by thread i, as that thread's k-th file. The same holds for the IV and the second stored field.*

The test never misses because a correct guess makes both sides the same deterministic function of the same inputs, which is Proposition 2. It never lies because Wrap is injective: OAEP encoding is injective for a fixed label and hash, and raising to the public exponent is a permutation of the integers modulo n. Two equal outputs therefore imply two equal inputs, so a wrong guess cannot match at all, rather than being unlikely to.

Checking the key wrap alone is enough. Checking both raises the agreement to 1024 bytes and gives a fallback when the first field has been damaged.

B. The decryption process

Our decryptor works like this.

Generate a table of keys and IVs using the timestamp as the seed, for each file and each thread, accounting for the number of logical processors. We have to assume any thread could have produced the key for any file, so we generate enough keys and IVs to cover every possibility across all files.

Encrypt the generated keys with the RSA public key pulled from the executable, the same way Rhysida does.

Compare that encrypted key against the value stored at the end of the encrypted file, which as a reminder is the encrypted key Rhysida stored there. If the values match, we have found the right key.

Decrypt the file with the recovered key. We only need to find the correct key for a single file to automatically decrypt all the others, because successfully decrypting one file means we have the right timestamp as the RNG seed. From there it is just a matter of testing each key in our table against the file we want to decrypt.

If we do not know the exact timestamp, we can start from an approximate one and sweep a time window around it, decrypting part of a file each time until we land on the exact timestamp.

C. Making the search linear

The table form just described costs one RSA operation per table entry, so $\mathcal{O}(PN)$ for a victim with N files.

The oracle runs in either direction. Instead of fixing a target file and searching the candidate space for it, fix a candidate and search the files. Build a hash map, once, from each file's stored wrapped key to the file itself, and a candidate is then tested with one lookup.

A thread's positions are consecutive. Thread i takes $k = 0, 1, 2, \ldots$ in order, and a file rejected by `isFileExcluded` consumes no stream at all, because every PRNG read happens inside `processFileEnc`. Marching k upward from zero therefore produces a run of hits that ends where that thread's work ended.

Algorithm 1 puts the two together. Phase 1 sweeps the whole window at $(i, k) = (0, 0)$ before trying any other position, so it costs one RSA operation per candidate second: thread 0's first file is some file in the victim set whenever thread 0 encrypted anything at all. The position is an output, not an input: the file that matches is thread 0's first. The position bound K and the thread bound P exist only to absorb the case where it did not. Phase 2 costs one RSA operation per recovered file, plus a miss budget of M consecutive failures per thread to detect where that thread's work ended, so $\mathcal{O}(N + MP)$ in total rather than $\mathcal{O}(PN)$.

With the whole file set in hand, the oracle's cost per candidate second does not depend on the file count at all: measured here at 0.32 ms throughout, against 156 ms per candidate for a 200 000-file table swept by trial decryption.

The set is often not in hand. A victim sends a handful of files, and thread 0's first file is then almost certainly not among them, so Phase 1 has to march (i, k) forward until it reaches a position one of the files held occupies. The number of steps is about N divided by one more than the number of files held. Measured on a 15 000-file run: 466 ms per candidate second with ten files held and 2.5 s with one, against 15 ms for the same sweep. The two forms cross over at a few hundred files *held*, whatever the run encrypted, since the crossover is the price of one RSA operation against one four-byte trial decryption. Below it the table form is faster, at the cost of a plaintext test that can return a false positive (Section VIII). Its two inputs matter again: the file count has to be an over-estimate, and a thread bound above the true count multiplies the steps, since the enumeration index is $kP + i$.

Phase 2 changes too. Its miss budget assumes the file set: with a handful of files almost every position is a miss, so M consecutive failures arrive at once and the phase stops before finding anything. Each thread has to be marched to the highest position held instead, a few seconds on a 15 000-file run at the measured rate.

Decryption itself is `ctr_decrypt` with the recovered key and IV over the same block layout the encryptor used (Figure 1), followed by truncating the 1036-byte trailer and dropping the `.rhysida` extension.

D. Finding the timestamp

`time(0)` is called at the very top of `main`, before the traversal starts, and every file is written after that. The earliest modification time among the `.rhysida` files on the victim is therefore an upper bound on t, and because the worker threads consume the traversal queue while the traversal is still running, it is a close one. A lower bound comes from any artefact of

Algorithm 1 Ciphertext-verified key recovery

Require: sample (giving n, e, L, H), the encrypted files, candidate window $\mathcal{W}$, thread bound P, position bound K, miss budget M
Ensure: seed t^* and a key for every recoverable file

$\mathcal{S} \leftarrow$ map from each file's stored wrapped key to that file
Phase 1: recover the seed
for $k \leftarrow 0$ **to** $K-1$ **do**
  for $i \leftarrow 0$ **to** $P-1$ **do**
    for all $t \in \mathcal{W}$ **do**
      $(key, iv, \sigma, \cdot) \leftarrow \mathrm{Gen}(t, i, k)$
      if $\mathrm{Wrap}(key, \sigma) \in \mathcal{S}$ **then**
        $t^* \leftarrow t$; **go to** Phase 2
      end if
    end for
  end for
end for
return failure
Phase 2: recover every file
for all $i \in [0, P)$ **do**
  $k \leftarrow 0$; $miss \leftarrow 0$
  while $miss < M$ **do**
    $(key, iv, \sigma, \cdot) \leftarrow \mathrm{Gen}(t^*, i, k)$
    if $\mathrm{Wrap}(key, \sigma)$ matches some $F \in \mathcal{S}$ **then**
      decrypt F with (key, iv); $miss \leftarrow 0$
    else
      $miss \leftarrow miss + 1$
    end if
    $k \leftarrow k + 1$
  end while
end for

the process starting: execution of the binary in the Windows event logs, prefetch, the USN journal, or the parent intrusion timeline. Between the two, the window is usually seconds to minutes.

The ransom notes and the replaced wallpaper carry write times from the same run.

Both routes depend on the file system metadata surviving. Restoring a backup over the encrypted tree, or copying the encrypted files off with a tool that does not preserve timestamps, leaves the files recoverable but widens the window from minutes to whatever can be bounded by other means.

E. What is needed for decryption

To decrypt a victim, we need the following.

The strain that infected the victim, since we need the RSA public key it contains.

The exact encryption timestamp. Or an approximate timestamp plus an encrypted file of known extension and type whose header we can guess (`.pdf`, `.docx`, etc.).

The approximate number of encrypted files, or an order of magnitude, in order to generate the key and IV table. This number must be greater than or equal to the number of encrypted files, never less. If unsure, give a big enough number: for 15 000 files actually encrypted, 200 000 is fine and 14 000 is not, and in that second case a thousand files will not be decrypted because the associated keys were never generated.

The number of logical processors on the encrypted machine.

In practice, the processor count and the number of encrypted files are easy to guess. The timestamp is the key piece.

TABLE III
INPUTS TO THE RECOVERY ONCE THE ORACLE DRIVES THE SEARCH, WITH THE VICTIM'S FILE SET IN HAND.

Input	Source, and what happens if it is wrong
The sample	Intrusion artefacts, a staging host, or open sources. Seven of the seventeen builds self-delete, so the sample is often no longer on the victim. Supplies n, e, L, H. Required.
Encryption timestamp, exact or a window	File system metadata on the `.rhysida` files, ransom note and wallpaper write times, event logs, prefetch, USN journal. A window costs one RSA operation per second in it.
Upper bound on logical processors	Victim hardware inventory, or simply 64. Too low loses the files handled by the missing threads. Too high is safe but not free: it costs M operations for each thread that never existed, and it must be raised together with the bound below.
Upper bound on files per thread	Total file count on the victim. Too low truncates each thread's run. Too high costs M extra operations per thread.

That is the list for the table-driven form. Two entries change once the oracle drives the search rather than only confirming its result, and Table III gives the revised list.

The known-header file drops out. The timestamp is still needed, but the file was only ever there to *check* a candidate timestamp, and Proposition 3 does that without decrypting anything.

With the file set in hand, the processor count becomes an upper bound rather than the exact number. With only a handful of files held, neither change applies and the table-driven list above is the one to use.

F. Reproducing the key schedule

Everything below was read off the sample and checked against a libtomcrypt build providing `chacha20_prng_desc`. A reimplementation has to link one that has it.

1) Use the C runtime's `rand`, not the host's. The samples are MinGW-w64 builds against the Microsoft C runtime, whose generator is $x_{j+1} = (214013\,x_j + 2531011) \bmod 2^{32}$ with output $(x_{j+1} \gg 16) \wedge \texttt{0x7fff}$ and $x_0 = t$.
2) Call `init_prng` $P+1$ times, not P. The first generator is never used for a file key and exists only to consume 41 draws. Thread i therefore consumes draws $[41(i{+}1),\ 41(i{+}2))$. The first 40 of them give $E_i[j] = (r_{41(i+1)+j} \cdot (idx + j + 1)) \bmod 256$, with $idx = 0$. The 41st is drawn and thrown away.
3) Register ChaCha20 and nothing else. idx above is a registry slot, so it is a property of the reimplementation rather than of the sample. Calling libtomcrypt's `register_all_prngs()` puts Yarrow, Fortuna and RC4 in first and returns 3 for ChaCha20, which changes every entropy byte and recovers nothing, with no diagnostic: the wrap never matches. Register `chacha20_prng_desc` alone, or hardcode $idx = 0$.
4) The ChaCha20 key and nonce are $G_i = \mathrm{KS0} \oplus E_i$, with KS0 the 40-byte constant given in Section V. Key $G_i[0..32)$, 64-bit nonce $G_i[32..40)$, counter 0, 20 rounds.

5) Discard 8 bytes of keystream, once, per generator. The length is a hard-coded immediate in `init_prng` and 8 is the value verified here, on the 64-bit builds. Read it out of the binary for any other target.
6) Per file, take 80 contiguous bytes: 32 key, 16 IV, 16 OAEP seed for the key wrap, 16 OAEP seed for the IV wrap.
7) OAEP with label `"Rhysida-0.1"` (11 bytes), hash and MGF1 both CHC over AES with $hLen = 16$, message length 32 for the key and 16 for the IV.
8) File decryption is AES-256-CTR through `ctr_start` with 14 rounds and libtomcrypt's little-endian counter mode, over the block layout of Figure 1, after stripping the 1036-byte trailer of Section IV.

Step 1 should be verified against the specific sample.

VII. Results

A. Reproducing the key schedule from the paper alone

The parameters in Section VI-F were implemented from scratch as a C program that links libtomcrypt: seed the `rand` generator with the candidate timestamp, run `init_prng` $P+1$ times, draw 80 bytes per file from the selected thread's stream, and wrap the key and the IV under the sample's embedded public key with the OAEP parameters given.

For every file in the corpus below, the program reproduces both 512-byte fields of the trailer, the wrapped key and the wrapped IV, byte for byte, with no access to any private key.

B. A worked example

The corpus is a set of ten files encrypted by Rhysida S0 on a two-processor machine, distributed with the decryptor as its regression test. The seed for that run is $t = 1687358453$. Take `jaas.conf.rhysida`, 4829 bytes on disk, so a payload of 3793 bytes and the 1036-byte trailer. It opens as in Listing 9.

```
c166123d32ee8b2ca93a456f7e71760f1e1321
68 7e667ee297a77e06d1aedb89 ...        (512 B)
00020000                               (len)
1571dab18e20413509be8ed4122df7667c468d
67 667114b4264723f53ef88a82 ...        (512 B)
00020000 01000000                      (len, tag)
```

Listing 9. The first bytes of each field of `jaas.conf.rhysida`'s 1036-byte trailer.

Seed the `rand` generator with t. Run `init_prng` three times: once for the standalone generator, then once for each of the two threads, consuming $3 \times 41 = 123$ draws in total. Advance thread 1's ChaCha20 stream past its 8-byte discard, then past four files at 80 bytes each, and read the next 80 bytes. The first 32 are the AES-256 key and the next 16 the IV:

```
key = 2f93a83a56bfa5acddcff0cbbf4c5d4e
      e869e0074f2cafcbfcd953afe938cecc
iv  = 369a13bd9ce66c430eb4fa05cba8ec28
```

The following 16 bytes are the OAEP seed for the key wrap. Feeding the key, that seed, the label `"Rhysida-0.1"` and the CHC-AES hash through OAEP and then through $x \mapsto x^e \bmod n$ returns exactly the 512 bytes the trailer opens with. The last 16 bytes of the 80 are the OAEP seed for the IV wrap, and they reproduce the second 512-byte field the same way.

TABLE IV
The worked example. Each file is located at a thread index i and a position k in that thread's stream. Blocks is the number of encrypted regions per Figure 1.

File	Payload (B)	Blocks	(i, k)
`NTUSER.DAT.LOG1`	73 728	1	(0, 0)
`mnemdb.json`	4 915 451	4	(0, 1)
`license.txt`	4 287	1	(0, 2)
`LaunchSupport.jar`	22 532	1	(0, 3)
`pythonRun`	1 646	1	(1, 0)
`Mono.Debugger.Soft.pdb`	62 328	1	(1, 1)
`lighthouse-develop.zip`	5 436 922	4	(1, 2)
`LGPL_3.0.html`	21 443	1	(1, 3)
`jaas.conf`	3 793	1	(1, 4)
`buildExtension.gradle`	11 319	1	(1, 5)

TABLE V
Measured cost of recovering the seed, one core. Every run places the true seed at the far end of the window, so the whole window is swept and the figures are worst case. Steady-state rate is 3 090 candidate seeds per second. The smallest window is dominated by process start-up.

Window	Seeds tried	RSA ops	Wall clock
±2 minutes	241	241	0.15 s
±1 hour	7 201	7 201	2.60 s
±24 hours	172 801	172 801	56.0 s

Decrypting is then AES-256-CTR over the payload, which here is below one block and so encrypted whole.

Table IV gives the full mapping for the ten files. The assignment of files to threads is scrambled relative to any traversal order. And thread 0's first file, the $(i, k) = (0, 0)$ entry that Phase 1 of Algorithm 1 depends on, is present in the set.

All ten decrypt correctly. `lighthouse-develop.zip` at four blocks exercises the whole layout of Figure 1, and after decryption all 128 of its members pass a CRC check, which would fail if one byte of the block or stride computation were wrong. The other four-block file, `mnemdb.json`, decrypts to valid text throughout, and the single-block `LaunchSupport.jar` passes CRC over its 16 members.

C. Cost of the seed search

Table V gives measured costs for Phase 1 of Algorithm 1 on one core of an Intel Haswell-generation vCPU under Debian 13, against the ten-file corpus. The RSA operation count is exactly one per candidate second.

Extrapolating at the measured rate, a one-month window is about 14 minutes on a single core and a one-year window about three hours. Even the degenerate case of no timestamp information at all is finite: the seed is a 32-bit value, so the whole space is 2^{32} candidates, roughly 16 core-days, and the loop has no shared state so it divides across cores without coordination.

Phase 2 located all ten files in 26 RSA operations, which is $N + MP$ for this corpus at a miss budget of $M = 8$.

D. The processor count is only an upper bound

Proposition 1 says the recovery needs a bound on the victim's logical processor count rather than its value. The run that produced the corpus used two threads. Repeating the recovery with the thread bound set to 2, 4, 8, 16 and 64 returns all ten files each time, at the same ten positions. Setting it to 1 returns only the four files that belong to thread 0.

Over-estimating is safe but not free. Every thread that never existed still has to be marched forward M positions before Algorithm 1 can conclude it did no work, so Phase 2 on this corpus costs 26 operations at the true bound of 2 and 522 at a bound of 64, which is $N + MP$ in both cases.

A second condition is a property of table-driven implementations rather than of the ransomware. A tool that pre-computes a table of $N \times P$ candidates but searches only its first N entries, with the thread index varying fastest, covers only N/P positions per thread. Raising P without raising N therefore shrinks the coverage per thread and starts losing files at high positions.

Both statements assume the file set is in hand (Section VI).

E. Coverage across strains

The same flaw persisted across the versions of Rhysida used by the attacking group, even though the program itself changed in various places. All seventeen samples of Table I carry it. Sixteen were confirmed against real encrypted files. S8, the one 32-bit build, was confirmed by static analysis alone. The changes between versions are in the traversal, the exclusion sets and the option handling. The key schedule is the same one in all of them: `srand(time(0))` at the top of `main`, `init_prng` $P + 1$ times, `chacha20_prng_read` for the key and IV, `rsa_encrypt_key_ex` with the same OAEP label.

The OAEP label `"Rhysida-0.1"` appears in all seventeen samples, and so does `"chc_hash"`, the name looked up to bind the OAEP hash to AES. By contrast the self-deletion command string appears in only seven of the seventeen, and the `"Number of procs"` debug message in only three, so the operator was editing around the key schedule without touching it.

The seventeen samples carry only twelve distinct RSA public keys. One key appears in four separate builds and two more appear in two each, so any build in the same key group carries the right public key.

Adapting the decryptor to each new version meant re-reading the RSA public key and re-locating the key schedule in a stripped binary.

F. Two builds the recovery does not reach

The Rhysida version we analysed that is written in PowerShell is not vulnerable to this decryption. The process for generating random numbers in that version does not use `srand` seeded with the timestamp, but a Windows function named `Security.Cryptography.RNGCryptoServiceProvider`.

The PowerShell encryptor was a second tool the group ran in parallel with the C/C++ one and used far less. The C/C++ encryptor kept shipping with `srand(time(0))` alongside it.

A later C/C++ build does resist the recovery. It was checked only far enough to establish that the recovery no longer works, and was not analysed to determine what changed or when the change was made.

We are not aware of any published analysis of the key generation of a post-2023 build, and the joint advisory's description of the encryption is unchanged in its April 2025 revision [1].

G. Limits

No sample. Without a strain carrying the right RSA public key there is nothing to re-wrap against. Decryption itself still only needs the seed, so what is lost is the exact test rather than the ability to recover: the fallback is to decrypt a few bytes and compare them against a known file-format signature, which is what the decryptor did in its fast path.

A damaged trailer. The oracle reads the last 1036 bytes of the file. If the file was truncated, or copied by something that mangled it, the candidate cannot be confirmed even when the key is correct, and the same plaintext fallback applies.

Windows only. Every sample in the corpus is a Windows PE. A Linux/ESXi Rhysida encryptor exists and has been publicly documented since late 2023.

Double encryption. A tree encrypted by two runs carries two trailers and two seeds. The search runs twice and the outer seed has to be found first.

The 32-bit build is untested end to end. S8 was confirmed vulnerable by reading it, not by recovering files from it. The length `init_prng` passes to `chacha20_prng_read` is a hard-coded immediate, 8 in the 64-bit builds, and a different value there would shift every subsequent byte of that thread's stream.

Intermittent encryption is not on this list. A file above one block is only partially encrypted (Figure 1), so the recovery restores the encrypted regions and the rest was never altered. The file comes back whole.

VIII. Comparison with Prior Public Work

A. Timeline

Avast published a technical analysis in October 2023 [12], KISA released a recovery tool in December 2023 [13], and Avast released one in February 2024 that it had been distributing to victims privately since August 2023 [14], which is the tool No More Ransom lists for the family [15]. The first public technical account is the preprint of Kim et al. in February 2024 [16], later published in extended form [4]. The analysis in this paper dates from May 2023.

B. Where the two accounts agree

The published account [4] reports the same MinGW build chain, the same exclusive use of libtomcrypt, the same ChaCha20 generator seeded through 40 bytes of `rand` output, the same $P + 1$ generator structure with the first one unused, the same AES-256-CTR with a little-endian counter, the same RSA-4096 wrap under OAEP, and the same 80 bytes of stream consumed per file.

TABLE VI
THE TWO RECOVERY METHODS, COMPARED ON WHAT EACH NEEDS AND WHAT EACH COSTS. N IS THE NUMBER OF ENCRYPTED FILES AND P THE THREAD COUNT. THE PROCESSOR COUNT AND THE TWO COST ROWS ASSUME THE VICTIM'S FILE SET IS AVAILABLE. SECTION VI GIVES WHAT CHANGES WHEN ONLY A FEW FILES ARE HELD.

	Prior work	**This work**
Confirms a key by	decrypting and judging the result	re-wrapping it and comparing 512 bytes
Plaintext knowledge	needed	none
False positives	possible	none
Decryption before confirmation	required	not required
Processor count	exact value	upper bound
Traversal order	reconstructed	not used
Modification times	required	used to bound the window
Cost per candidate seed	$\mathcal{O}(P)$ trial decryptions	1 RSA operation
Cost to place all files	$\mathcal{O}(PN)$	$\mathcal{O}(N + MP)$

C. Where they differ: confirming a candidate

The published method decrypts the first file encrypted by each thread and concludes the seed is correct when one or more of them decrypt correctly, which leaves the test resting on whatever criterion decides that a decryption looks right. The table form of Section VI had the same property in its fast path, where it compared four decrypted bytes against a known file-format signature and could return a false positive once the swept window grew large.

The oracle answers without decrypting anything. Both accounts observe that the OAEP seeds come out of the regenerable generator. The step taken here is to use the attacker's own 512-byte ciphertext as a reproducible self-check, and to build the search on it. Table VI sets out what changes as a result.

IX. DISCUSSION

The seeding bug is the obvious one. `srand` and `time` sit in the import table of the sample, and a triage step that greps for that pair tells you whether a family is worth attacking at all.

The second failure is harder to spot. libtomcrypt's `rsa_encrypt_key_ex` takes a `prng_state` and a PRNG index as parameters, because OAEP has to draw a seed and the library will not pick a source for the caller. The API gives no signal about the quality that source needs to have. Rhysida passes it the state it just used to produce the message, which is the natural thing to do if the PRNG is thought of as "the program's random number generator" rather than as part of the security argument.

In a hybrid scheme where one predictable generator feeds both the session key and the randomised padding of the key wrap, the wrapped-key blob stops being opaque and becomes a key-confirmation oracle for anybody who can guess the generator's state.

X. ETHICS, DISCLOSURE AND AVAILABILITY

A. Disclosure

There is no vendor to notify here.

Publishing now costs the operator nothing they have not already lost. The underlying flaw has been public since the preprint of Kim et al. [16] in February 2024, and a later C/C++ build already resists the recovery altogether (Section VII-F). Nothing in this paper gives an attacker a capability they lack: whoever holds the RSA private key can already decrypt anything they encrypted.

B. Availability

We have chosen not to publish the decryptor.

The vulnerability is now well documented and free decryptors built by other parties are already available to victims through nomoreransom.org [15].

Section VI-F gives every constant and every ordering decision needed to rebuild the key schedule, so a reader with a sample can reimplement the recovery from that section alone and check our claims against their own copy of the binary.

C. Handling of victim data

No victim data appears in this paper. The only victim organisation named anywhere is the British Library, cited solely to its own published incident review. Fifteen of the seventeen sample hashes are disclosed, and all fifteen were already documented in open sources before this paper. The two that were not are withheld, because a hash of a sample tied to a specific incident is itself an identifier of that incident. The ten encrypted files used for the worked example in Section VII-B are the decryptor's own regression corpus and contain no victim material.

D. Dual use

The analysis recovers files that a ransomware operator encrypted, using material that operator left in the victim's own files. It does not generalise to families that seed their generators correctly.

XI. RELATED WORK

The line of work closest to this one recovers victim files by breaking the ransomware's key management rather than its cipher. Kim et al. recovered Hive by observing that the master key is only partially consumed, so 95% of it can be reconstructed from the encrypted files without the RSA private key [17]. Lee et al. broke Magniber v2 by attacking a generator the author wrote by hand instead of using a library one [18].

The nearest precedent for the specific defect is LooCipher, one of the five families in Lee et al.'s 2019 study [19], which seeds `rand` from the system time and is recoverable by searching the interval between key generation and the file modification time. That is the same shape of attack as the one here, four years earlier and on a different family. Earlier still, Linux.Encoder derived its key and IV from `rand` seeded with the system timestamp at the moment of encryption, recoverable directly from the file's own timestamp [20]. Seeding

a cryptographic generator from the clock is catalogued as CWE-337 [21].

On Rhysida specifically, the joint FBI, CISA and MS-ISAC advisory AA23-319A is the reference document for the group's tradecraft and indicators [1], though its account of the encryption differs from what the binary does: it describes "a 4096-bit RSA encryption key with a ChaCha20 algorithm", where in the binary ChaCha20 is the pseudo-random generator and file data is encrypted with AES-256 in CTR mode. Avast published a technical analysis that correctly identifies the ChaCha20 generator as the source of the AES key, the IV and the OAEP padding [12]. Kim et al. are the published academic treatment [4].

XII. Conclusion

Rhysida encrypted files under AES-256 in CTR mode and wrapped every key with a 4096-bit RSA public key. The construction was correct. It was built on `srand(time(0))`, and one 32-bit integer that can be read off the victim's own file system determines every key the run produced.

Rhysida handed the same broken generator to `rsa_encrypt_key_ex`, so the OAEP seed came out of the same stream as the key it was padding. The RSA blob the encryptor appends to every file became a byte-for-byte reproducible check on any guess.

The flaw held across all seventeen variants analysed. It is absent from a later C/C++ build, and from the group's PowerShell encryptor, which was never vulnerable.

Over the course of this work, dozens of victims around the world had their files recovered and their data saved, without ever paying a ransom.

References

[1] Federal Bureau of Investigation, Cybersecurity and Infrastructure Security Agency, and Multi-State Information Sharing and Analysis Center, "#StopRansomware: Rhysida ransomware," Cybersecurity and Infrastructure Security Agency, Tech. Rep. AA23-319A, Nov. 2023, released 15 November 2023; last revised 30 April 2025. [Online]. Available: https://www.cisa.gov/news-events/cybersecurity-advisories/aa23-319a

[2] Ransomware.live, "Rhysida group profile," https://www.ransomware.live/group/rhysida, 273 listed victims across 39 countries, measured 1 August 2026.

[3] The British Library, "Learning lessons from the cyber-attack: British Library cyber incident review," The British Library, Tech. Rep., Mar. 2024, published 8 March 2024, 18 pp. [Online]. Available: https://www.bl.uk/home/british-library-cyber-incident-review-8-march-2024.pdf

[4] G. Kim, S. Kang, S. Baek, K. Kim, and J. Kim, "How to decrypt files encrypted by Rhysida ransomware without the attacker's private key," *Computers & Security*, vol. 151, p. 104340, Apr. 2025.

[5] T. S. Denis, *LibTomCrypt: A Comprehensive, Modular and Portable Cryptographic Toolkit*, public domain. [Online]. Available: https://github.com/libtom/libtomcrypt

[6] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," RFC 8439, Jun. 2018.

[7] National Institute of Standards and Technology, "Advanced encryption standard (AES)," National Institute of Standards and Technology, Tech. Rep. FIPS PUB 197, Update 1, May 2023.

[8] M. J. Dworkin, "Recommendation for block cipher modes of operation: Methods and techniques," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-38A, 2001.

[9] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #1: RSA cryptography specifications version 2.2," RFC 8017, Nov. 2016.

[10] ISO/IEC JTC1/SC22/WG14, "Programming languages — C," ISO/IEC, Tech. Rep. ISO/IEC 9899:2024, working draft N3220, 2024, clauses 7.24.2.1 and 7.24.2.2. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf

[11] A. Milenkoski and J. Walter, "Crimeware trends: Ransomware developers turn to intermittent encryption to evade detection," SentinelLabs, Sep. 2022. [Online]. Available: https://www.sentinelone.com/labs/crimeware-trends-ransomware-developers-turn-to-intermittent-encryption-to-evade-detection/

[12] Gen Digital Threat Research Team, "Rhysida ransomware technical analysis," https://www.gendigital.com/blog/insights/research/rhysida-ransomware-technical-analysis, Oct. 2023.

[13] Korea Internet and Security Agency, "Rhysida ransomware decryption tool, version 1.0," https://seed.kisa.or.kr/kisa/Board/166/detailView.do, Dec. 2023, released 21 December 2023.

[14] Gen Digital Threat Research Team, "Decrypted: Rhysida ransomware," https://www.gendigital.com/blog/insights/research/decrypted-rhysida-ransomware, Feb. 2024.

[15] No More Ransom, "The no more ransom project," https://www.nomoreransom.org/, launched 25 July 2016 by the Dutch National High Tech Crime Unit, Europol's European Cybercrime Centre, Kaspersky and McAfee.

[16] G. Kim, S. Kang, S. Baek, K. Kim, and J. Kim, "A method for decrypting data infected with Rhysida ransomware," arXiv:2402.06440 [cs.CR], Feb. 2024.

[17] G. Kim, S. Kim, S. Kang, and J. Kim, "A method for decrypting data infected with Hive ransomware," *Journal of Information Security and Applications*, vol. 71, p. 103387, Dec. 2022.

[18] S. Lee, M. Park, and J. Kim, "Magniber v2 ransomware decryption: Exploiting the vulnerability of a self-developed pseudo random number generator," *Electronics*, vol. 10, no. 1, p. 16, 2021.

[19] S. Lee, B. Youn, S. Kim, G. Kim, Y. Lee, D. Kim, H. Park, and J. Kim, "A study on encryption process and decryption of ransomware in 2019," *Journal of the Korea Institute of Information Security and Cryptology*, vol. 29, no. 6, pp. 1339–1350, Dec. 2019.

[20] B. Botezatu, "Linux ransomware debut fails on predictable encryption key," Bitdefender Labs, Nov. 2015, https://www.bitdefender.com/en-us/blog/labs/linux-ransomware-debut-fails-on-predictable-encryption-key.

[21] MITRE, "CWE-337: Predictable seed in pseudo-random number generator (PRNG)," Common Weakness Enumeration, https://cwe.mitre.org/data/definitions/337.html.

Appendix A
Exclusion Sets

Both sets are stored in the binary as integer arrays, one character per element, padded to a fixed stride: 33 for the directory paths and 12 for the extensions.

A. Excluded directories

Eleven entries, checked by `isDirectoryExcluded`.

```
/$Recycle.Bin                /Recovery
/Boot                        /System Volume Information
/Documents and Settings      /Windows
/PerfLogs                    /$RECYCLE.BIN
/Program Files
/Program Files (x86)
/ProgramData
```

Both spellings of the recycle bin appear, and the comparison is case-sensitive.

B. Excluded extensions

Twenty-seven entries, checked by `isFileExcluded`.

```
.bat    .cur     .drv    .ico    .ps1    .ini
.bin    .diagcab .dll    .lnk    .psm1   Thumbs.db
.cab    .diagcfg .exe    .msi    .scr    .url
.cmd    .diagpkg .hlp    .ocx    .sys    .iso
.com             .hta                    .cab
```

`.cab` appears twice in the array, at positions 3 and 27. The entry at position 24 is `Thumbs.db`, a full file name rather than an extension.

Appendix B
Additional Listings

```
1  void __cdecl parseOptions(int argc, char **argv, Options
       *options)
2  {
3    size_t v3;                    // rbx
4    char _selfremoved[24];        // [rsp+20h] [rbp-60h]
     BYREF
5    char self_remove_modifier[4]; // [rsp+41h] [rbp-3Fh]
     BYREF
6    char directory_modifier[3];   // [rsp+45h] [rbp-3Bh]
     BYREF
7    int dir_n;                    // [rsp+48h] [rbp-38h]
8    int i;                        // [rsp+4Ch] [rbp-34h]
9
10   options->program   = (char *)malloc(0x1000ui64);
11   options->directory = (char *)malloc(0x1000ui64);
12   *options->directory = 0;
13   options->is_self_remove = 1;
14   strcpy(directory_modifier, "-d");
15   strcpy(self_remove_modifier, "-sr");
16   for ( i = 0; i < argc; ++i )
17   {
18     if ( i )
19     {
20       if ( !strcmp(argv[i], directory_modifier) )
21       {
22         if ( argv[++i] )
23         {
24           strcpy(options->directory, argv[i]);
25           for ( dir_n = 0; ; ++dir_n )
26           {
27             v3 = dir_n;
28             if ( v3 >= strlen(options->directory) )
29               break;
30             if ( options->directory[dir_n] == '\\' )
31               options->directory[dir_n] = '/';
32           }
33         }
34       }
35       else if ( !strcmp(argv[i], self_remove_modifier) )
36       {
37         strcpy(_selfremoved, "I'm will be selfremoved");
38         puts(_selfremoved);
39         options->is_self_remove = 1;
40       }
41     }
42     else
43     {
44       strcpy(options->program, *argv);
45     }
46   }
47 }
```

Listing 10. `parseOptions` in full. `options->is_self_remove` is set to 1 on line 13, before any argument is looked at.

Backslashes in a `-d` argument are rewritten to forward slashes, which is why the exclusion paths in Appendix A are written with forward slashes despite targeting Windows.

```
1  if ( options->is_self_remove == 1 )
2  {
3    command = (char *)malloc(0x7FFui64);
4    strcpy(command,
5      "cmd.exe /c start powershell.exe -WindowStyle Hidden
       "
6      "-Command Sleep -Milliseconds 500; "
7      "Remove-Item -Force -Path \"");
8    strcat(command, cwd);
9    *(_WORD *)&command[strlen(command)] = '\\';
10   strcat(command, options->program);
11   strcat(command, "\" -ErrorAction SilentlyContinue;");
12 }
13 ...
14 if ( options->is_self_remove == 1 )
15 {
16   system(command);
17   free(command);
18 }
```

Listing 11. The self-deletion routine in S0, reached at the end of a run.

`cwd` is the working directory captured by the `getcwd` call on line 3 of Listing 2, and `options->program` is `argv[0]`. The path is therefore only correct when the encryptor was launched with a relative path from its own directory.

Appendix C
Sample Hashes

Full SHA-256 values for the fifteen samples of Table I that are already documented in open sources, either in the sample table of Kim et al. [16] or on MalwareBazaar. The **Key** column groups samples sharing an embedded RSA public key: K1 covers S0, S4, S5 and S16, K2 covers S1 and the withheld S9, and K7 covers S8 and S12.

ID	SHA-256	Key
S0	a864282fea5a536510ae86c77ce46f78 27687783628e4f2ceb5bf2c41b8cd3c6	K1
S1	d5c2f87033a5baeeb1b5b681f2c4a156 ff1c05ccd1bfdaf6eae019fc4d5320ee	K2
S3	3518195c256aa940c607f8534c91b5a9 cd453c7417810de3cd4d262e2906d24f	K4
S4	67a78b39e760e3460a135a7e4fa096ab 6ce6b013658103890c866d9401928ba5	K1
S5	250e81eeb4df4649ccb13e271ae3f80d 44995b2f8ffca7a2c5e1c738546c2ab1	K1
S6	a78fa1ecadc46870c17e458ab427bad6 586b74c7d3e8472f6d8448832ccb20f1	K5
S7	258ddd78655ac0587f64d7146e525491 15b67465302c0cbd15a0cba746f05595	K6
S8	3d2013c2ba0aa1c0475cab186ddf3d90 05133fe5f88b5d8604b46673b96a40d8	K7
S10	2c5d3fea7ad3c9c49e9c1a154370229c 86c48fbaf7044213fd85d31efcebf7f6	K8
S11	f6f74e05e24dd2e4e60e5fb50f73fc72 0ee826a43f2f0056e5b88724fa06fbab	K9
S12	0bb0e1fcff8ccf54c6f9ecfd4bbb6757 f6a25cb0e7a173d12cf0f402a3ae706f	K7
S13	8061bb999a0f5d3165742283001a7a68 e7905718c928172343bf8456b69f268d	K10
S14	1a9c27e5be8c58da1c02fc4245a07831 d5d431cdd1a91cd35d2dd0ad62da71cd	K11
S15	b55ecbddcbed916481ad537807cd3e33 cb71814be6ce8e03eb63b629ccb8c692	K12
S16	0050a69d6e93eddcd1ea4b7e951945f8 970e5700d9436238bde7f63d757988ae	K1