Bachelor's Thesis Computing Science

Radboud University Nijmegen

The disappearance of constant-time execution.

A C compiler's magic trick.

Author:
Vincent Dankbaar
s1031350

First supervisor/assessor:
Professor Peter Schwabe

Second assessor:
Professor Lejla Batina

March 31, 2024

Abstract

In cryptography implementation contexts certain compiler optimizations can be undesirable. It is possible for various versions of the Clang compiler, using varying optimization levels, to turn constant-time source code into variable-time binaries. By testing various cryptographic implementations with a tool called TIMECOP we produce some interesting results. Most notably that Clang versions 17+ produce variable-time binaries with optimization level -O3 enabled where older versions did not. Additionally, -Os optimization leads to variable-time binaries also on older versions of Clang. We also show that a NTRU Prime primitive called `sntrup761`, which is currently part of the default configuration for OpenSSH, is vulnerable to this undesired optimization behaviour.
Keywords: Constant-time, compiler, optimization, side-channel, timing attack

Contents

Chapter 1

Introduction

Many programs used in everyday life depend on the various cryptography schemes created by the cybersecurity community. A lot of effort goes into proving these schemes are theoretically secure, in as far as that is possible. However, what works in theory sometimes breaks down when it comes to implementation. *Side-channel attacks* are a type of attack with which attackers can learn information about the keys and other secrets used in a scheme, not because the scheme itself is weak but because the implementation or the hardware it runs on leaks information that can be used.

One specific type of side-channel attack is called a *timing attack*. When a cryptography implementation is vulnerable to a timing attack, that means that an attacker could recover secret information by measuring the execution time of the implementation.

To prevent these timing attacks, cryptography implementations are written following the so-called "constant-time" programming paradigm. This way of programming decouples the execution time from the secret information. However, some projects include general-purpose compilers as intermediaries, where constant-time source code is compiled to produce binaries. This thesis explores the risk of such a compiler behaving in unexpected ways. Specifically when the compiler optimizes away the programmer's effort of making their source code constant-time, instead producing a variable-time binary.

Previous works have shown that this behaviour can occur. In this thesis, we test various versions of the LLVM Clang compiler using TIMECOP to show how this undesirable behaviour has progressed over time. All tests are executed with two different optimization levels and also with no optimization as a control. The tests that pass without optimization but fail with optimization are of interest. The main goal is to see if there is a correlation between the compiler version and the number of failures. As well as see if

there are any notable implementations of cryptographic schemes that would be affected by this undesirable optimization behaviour.

First, we give an overview of the previous work that has been done on this problem to show that this behaviour is possible at all and to introduce the various existing methods and tools that have been developed to counter timing attacks in general. Next, we look at the methods used in this thesis and how the results were processed. Then we present the results and show how they were further utilized. Finally there is a discussion on the findings and a summarized conclusion.

We conclude that there is correlation that the newer the compiler version is, the more constant-time violations there are. These violations could open the way for timing attacks on the cryptographic implementations and subsequently on software that uses those implementations. To further demonstrate the risk, we show that some of the code that failed our tests is part of the NTRU Prime project which is currently used as a default for key exchange in OpenSSH.

Chapter 2

Related Work

Below we dig into some of the work that has already been done on the topic of timing attacks. Starting with a look what causes timing attacks to be possible and some examples of timing attacks in practice to show why they are relevant and pose a threat. We then elaborate on the research that has been done to detect and prevent timing attacks. Finally we highlight some of issues with detecting an preventing timing attacks since the discovery of the Spectre family of attacks.

Timing attacks

A 1996 paper by Kocher [20] was the first to highlight timing attacks as a significant threat. Mainly due to the potential of recovering full secret keys from the Diffie-Hellman, RSA and DSS implementations that were in use at that time.
Since that first publication timing attacks have become a well researched topic. The constant-time programming paradigm has become a standard to reduce their effect and over time, papers have been written highlighting the problem from different angles.

The cause of timing attacks

Whether or not a timing attack is possible depends on the implementation. Below we list some of the common underlying reasons why timing attacks could be possible for a given implementation.

In branching code, often one branch takes longer to execute than the other. If secret data decides which branch gets executed then, by measuring the execution time, information can be gained about that secret data.

Variable-time processor instructions are another clear cause. Multiplication and division instructions are common causes of timing attacks due to

their often non constant-time implementation under the hood.

Finally indexing based on secret data regularly leads to timing attacks. As Bernstein wrote in his 2005 paper on cache-timing attacks on AES: "Using secret data as an array index is a recipe for disaster" [7]. When using secret data as an index, the CPU tries to load the data from cache, sometimes this hits, meaning the data was found in the cache, and sometimes it misses, in which case it needs to be fetched from RAM which takes longer. The address used in the lookup is secret itself because it is calculated using the secret data. Cache timing-attacks make use of these facts. By using various techniques, for example FLUSH+RELOAD [24], attackers prepare the cache in certain ways and keep track of cache usage to determine the value of the secret data.

Examples of timing attacks

There are many papers that show timing attacks in practice. Both in hardware and software. They serve to highlight the significance and potential threat that timing attacks pose.

In a paper by Kaufmann et al. [18] a constant-time implementation of Elliptic Curve Diffie Hellman (ECDH) Curve25519 was compiled with MSVC 2015. The resulting binary relied on a windows run-time library which contained a variable-time multiplication. This then caused the binary to be vulnerable to a timing attack. In the paper they go on to give an example of how, by measuring the execution time, they can recover the secret key.

A paper by Pereida et al. [21] describes how a cache-timing attack could be used to recover the secret key from the OpenSSL's DSA implementation, affecting both TLS and SSH which make use of OpenSSL.
Another paper by Yarom et al. [25] attacks OpenSSL's RSA implementation. They demonstrate how to recover a RSA private key by using a cache-timing attack called CacheBleed.

A 2005 paper by Brumley and Boneh [9] demonstrates that it is possible to perform timing attacks remotely. They demonstrate how to recover a private key from OpenSSL's RSA implementation between two processes on the same machine, between two virtual machines and over network.

Preventing timing attacks

Obviously timing attacks are not desirable, so a lot of research has been done on how to detect and prevent them. Most aim to provide developers with

new tools to detect the weaknesses and new languages and compilers that help to prevent them. Although many developers are aware of the threat that timing attacks pose, it turns out that the number of developers that actually make use of the tools available, is lower than desirable [17]. The main causes seem to be that these tools are hard to use, lack integration with existing workflows and that the outputs of such tools can be hard to interpret.

Detection

The existence of timing attacks has been known for a long time now. As such, there do exist a lot of tools for detecting them.
A paper by Fourne et al. [16] provides a nice overview of the various tools for verifying constant-time behaviour. It also lists whether these tools provide any guarantees and a rating of their usability. As highlighted in the paper, the main problem with many of the tools is that the testing process is rather involved. Using them often requires adjustments to the existing code and knowledge about how to correctly set up and use the tools. Below you can see a subset of the tools from this list and a simple description of how they work.

TIMECOP is a dynamic analysis tool that works by 'poisoning' the secret data. This poisoning basically marks the secret data as uninitialized in the eyes of the Valgrind memory checker. Valgrind can then be used to check for variable-time behaviour as it will report whether that uninitialized data is being used to perform any conditional branching or indexing.

Binsec-rel [13] [14] can be used to test binaries for variable-time execution. It works by combining various 'state-of-the-art techniques such as binary-level formal methods, symbolic execution, abstract interpretation, SMT solving and fuzzing'.

Dude-CT [22] is a tool that helps automate timing for various inputs and then uses statistics to determine whether the function is constant-time or not.

Pitchfork [15] is a symbolic analysis tool. It is able to test LLVM bitcode for constant-time execution. To use it the developer needs to write a wrapper for the function they want to test in rust.

Prevention

One way to try to prevent the generation of variable-time binaries is by using a compiler that is guaranteed to preserve constant-time execution even after

optimization. A paper by Barthe et al. [6] provided mathematical proofs of correctness in Coq which were then applied to modify CompCert [4], which is a compiler fully verified with Coq. The modifications lead to a version of CompCert proven to preserve constant-time execution when compiling and optimizing constant-time C code.

Another way to solve this problem is to make a new language entirely. These types of languages are called DSLs or Domain-Specific Languages. Below are two examples of such languages.

Jasmin [1] is a programming language created with security and speed in mind. Its included compiler has been fully verified using Coq, which then gives guarantees about whether the compiled binaries will be constant-time, even after optimization.
Jasmin is built in a way to allow both low-level control as well as high-level abstractions while keeping the resulting binaries constant-time.
In some subsequent papers [2] [3] Jasmin is used to produce safe binaries with high performance.
That Jasmins compiler preserves constant-time behaviour was also proven in a paper by Barthe et al. [5].

In a similar spirit Cauligi et al. [12] developed a special language called FaCT which they describe as 'a high-level, strongly-typed C-like DSL(Domain-Specific Language)'. Instead of the lower-level approach that Jasmin takes, FaCT is higher level which allows it to 'automatically apply the recipes that developers have hitherto applied by hand' that make up constant-time C code.

Other DSL's and interesting work include Vale [8] for 'formally verified high-performance assembly', HACL* [26] (made with Low*) which is a 'verified portable C cryptographic library' and CT-wasm [23] which is an extension of Web-Assembly that helps with writing constant-time cryptography implementations for the web.

Spectre

Although writing cryptographic implementations in a constant-time way definitely helps to prevent timing attacks, it is not perfect. Since the Spectre [19] family of attacks, one can no longer fully rely on the fundamental assumptions that formed the backbone of constant-time programming. 'The decade-old constant-time recipes are no longer enough' [10]. 'Spectre attacks—and speculative execution in general—violate our typical assumptions and abstractions and have proven particularly challenging to reason about and defend against' [11]. Even now, work is still being done on build-

ing mitigations, on hardware and software level, and writing patches to prevent Spectre attacks and other micro-architectural attacks that exploit speculative execution.

Chapter 3

Methodology

The compilers

In this paper we look specifically at the behavior of x86_64 Clang versions on Debian. Other compilers and versions were not taken into account yet due to the long testing times; however the same methodology could be applied with a different compiler like gcc or other compiler versions.
We test the versions of Clang mentioned below. All of these versions are relatively new with 15.0.0 being released in September 2022 and 18.1.1 most recently in March 2024. These versions were chosen as they give a good indication of how fast security-relevant compiler behaviour can change. We test the most recent versions as they likely use the newest high-level optimizations which could produce the unwanted optimization behaviour we are looking for.

Clang versions
18.1.1
17.0.6
17.0.2
16.0.4
16.0.0
15.0.6
15.0.0

The following three sets of compilation flags were used.

```
-march=native -g -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE
-march=native -g -O3 -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE
-march=native -g -Os -fomit-frame-pointer -fwrapv -Qunused-arguments -fPIC -fPIE
```

Most of these flags come from default flags that SUPERCOP uses for Clang compilation. The flags added that are of interest are:

- -O3: optimizes for speed as much as is possible while staying C standard compliant.
- -Os: optimizes to make the storage size of the binary as small as possible.
- -g: adds debugging symbols, this allows TIMECOP to refer to specific program lines in its fails.

After a SUPERCOP update in 2024, the '-lgmp' flag is also needed to compile everything correctly. Some tests were done before and some after this change. Refer to the raw data in the github listed in the appendix to see exactly with which flags each result was obtained.

SUPERCOP & TIMECOP

TIMECOP as mentioned in the related works is a dynamic analysis tool for finding constant-time violations. It is the main tool constant-time verification tool used in this paper. Since its release, TIMECOP has been merged into the SUPERCOP suite.
SUPERCOP is a tool that is normally used to benchmark implementations of various cryptographic primitives. However, It is possible to run the full SUPERCOP suite with TIMECOP enabled, this way each implementation can be tested automatically with the compiler configuration the user provides.
See the appendix for a direct link to the TIMECOP and SUPERCOP website.

After downloading SUPERCOP we edit okcompilers/c to have only the lines for the desired versions and flags and remove all other entries from okcompilers/cpp and okcompilers/rs.
When adding the compiler entries to SUPERCOP, the Clang versions must be referenced with an absolute path or it will not work correctly. For example, after adjusting the exact path, the following could work correctly as an entry in okcompilers/c for SUPERCOP:

```
/home/user/clang/17.0.6/bin/clang-17 -march=native -g -fomit-frame-pointer
    -fwrapv -Qunused-arguments -fPIC -fPIE -lgmp
```

Runing the tests

To replicate the results in this paper, execute:

```
env TIMECOP=1 ./do-part used
```

This makes SUPERCOP only run tests on the 'subroutine' implementations that are used in other implementations, saving a lot of time. A full run of the SUPERCOP suite with only one compiler entry could take up to 2 days according to the official site. For comparing various Clang versions like in this paper this would be unnecessarily long.

Some notes

SUPERCOP did not run well when in a Docker container or VM, part of the issue seemed to be that it requires kernel access for some of its measurements. I was hoping to speed up testing by using virtualization but in the end SUPERCOP was most reliable when running directly on the host OS. It did work in WSL2 on Windows but it was slower.

SUPERCOP allows for some interesting parallelization to reduce test time by running the tests on multiple cores instead of just one. However, this did not seem to work with TIMECOP. When used SUPERCOP only runs the benchmarks without running TIMECOP. So be sure to not use SUPERCOP's 'data' programs, but only the 'do' or 'do-part' programs.

The combination of the above makes running the tests very slow. For a single Clang version with the 3 flag variations it took about a full day on a laptop with an AMD Ryzen 7 5800H. Take this into account when planning extensive testing. I ran some subsets of the version and flag combinations simultaneously on different laptops to speed up the process, since only the TIMECOP results would be relevant and not the benchmarks.

Result collection

After completion of the tests, a data file will be generated in the 'bench' folder of SUPERCOP. This file contains in plain text all the results of the tests. The TIMECOP results are mixed in with the benchmark results. Also, each compiler configuration entry will be done in order before continuing to the next test. This makes it a bit hard to read and draw conclusions from the output.
To aid in the extracting of the interesting results I wrote some simple Python scripts. These can be found on the GitHub repository listed in the appendix. One generates a system of folders where each output of a TIMECOP test is listed separately by Clang version and flag. The other simply extracts all the TIMECOP fails, which can then be piped to a csv file.

Chapter 4

Results

After running all the tests and parsing the resulting data, we obtain the following results.

Clang version	**Timecop pass**	**Timecop error**	**Timecop fail**
18.1.1	1615	20	30
17.0.6	1615	20	30
17.0.2	1609	20	30
16.0.4	1616	21	28
16.0.0	1616	21	28
15.0.6	1624	21	20
15.0.0	1624	21	20

Table 4.1: TIMECOP Pass/Error/Fail

Small note: 6 tests have been left out from the results of version 17.0.2 above.
These are the following:

- crypto_hash/blake512/sse2: O3 and Os
- crypto_hash/blake512/sse2s: O3 and Os
- crypto_hash/blake512/ssse3: O3 and Os

These tests were manually skipped as SUPERCOP had frozen while running them.

TIMECOP fails by optimization

The table below shows how many TIMECOP fails were caused by each flag and in which version:

Clang version	O3	Os	No optimization
18.1.1	26	4	0
17.0.6	26	4	0
17.0.2	26	4	0
16.0.4	24	4	0
16.0.0	24	4	0
15.0.6	18	2	0
15.0.0	18	2	0

Table 4.2: TIMECOP fails by flag

List of failing subroutine implementations

Below are the names of the 'used' subroutine implementations that failed any TIMECOP tests, which category they belonged to and how many times they failed on TIMECOP:

Catagory	Subroutine	Implementation	Fails
crypto_core	invhrss701	ref	10
crypto_core	rainbowcalsecret	amd64	21
crypto_core	rainbowcalsecret	ref	21
crypto_core	wforcesntrup	ref	42
crypto_core	wforcesntrup	ref2	42
crypto_core	wforcesntrup	simpler	30
crypto_scalarmult	curve25519	ref	10
crypto_scalarmult	kummer	ref	5
crypto_scalarmult	kummer	ref5u	5

Table 4.3: TIMECOP fails by implementation

TIMECOP output & source code

For each entry below, it is listed which implementation failed under which flags. If there are multiple 'versions' of the same implementation then these are also listed.

invhrss701

The 'ref' implementation failed TIMECOP.
Flag 'Os' failed on all Clang versions:

```
==4433== Use of uninitialised value of size 8
==4433== at 0x10AD27: cmov (core.c:69)
==4433== by 0x10AD27: crypto_core_invhrss701_ref_constbranchindex
    (???:141)
==4433== by 0x109437: test (try.c:106)
==4433== by 0x109FE1: main (try-anything.c:345)
==4433== Uninitialised value was created by a client request
==4433== at 0x109E9F: poison (try-anything.c:281)
==4433== by 0x1093FA: test (try.c:103)
==4433== by 0x109FE1: main (try-anything.c:345)
```

Flag 'O3' failed on 17.0.2, 17.0.6 and 18.1.1:

```
==32707== Conditional jump or move depends on uninitialised value(s)
==32707== at 0x10D0E0: cmov (core.c:69)
==32707== by 0x10D0E0: crypto_core_invhrss701_ref_constbranchindex
    (???:141)
==32707== by 0x10947B: test (try.c:106)
==32707== by 0x10A73E: main (try-anything.c:345)
==32707== Uninitialised value was created by a stack allocation
==32707== at 0x10B061: crypto_core_invhrss701_ref_constbranchindex (core
    .c:73)
```

The 'cmov' function mentioned in the output looks like follows.
According to TIMECOP line 6 caused the fail.

```
static void cmov(unsigned char *r, const unsigned char *x, size_t len,
    unsigned char b)
{
  size_t i;
  b = -b;
  for(i=0;i<len;i++)
    r[i] ^= b & (x[i] ^ r[i]);
}
```

rainbowcalsecret

Both the ‘amd64’ and ‘ref’ implementations of the ‘363232’, ‘683248’ and ‘963664’ versions failed on TIMECOP.
Flag ‘O3’ failed on all Clang versions:

```
==22257== Conditional jump or move depends on uninitialised value(s)
==22257== at 0x10CB33: gf4v_mul_u32 (gf16.h:52)
==22257== by 0x10CB33: gf16v_mul_u32 (gf16.h:129)
==22257== by 0x10CB33: gf256v_mul_u32 (gf16.h:265)
==22257== by 0x10CB33: _gf256v_madd_u32 (blas_u32.h:129)
==22257== by 0x10DC18:
    crypto_core_rainbowcalsecret683248_ref_constbranchindex
_batch_2trimat_madd_gf256(parallel_matrix_op.c:131)
==22257== by 0x10B969: calculate_F_from_Q_impl (
    rainbow_keypair_computation.c:243)
==22257== by 0x10B969: _calculate_F_from_Q (???:406)
==22257== by 0x10B969:
    crypto_core_rainbowcalsecret683248_ref_constbranchindex (???:416)
==22257== by 0x1094B2: test (try.c:106)
==22257== by 0x10A802: main (try-anything.c:345)
==22257== Uninitialised value was created by a client request
==22257== at 0x10A628: poison (try-anything.c:281)
==22257== by 0x109483: test (try.c:104)
==22257== by 0x10A802: main (try-anything.c:345)
```

The ‘gf4v_mul_u32’ function mentioned in the output looks like follows. According to TIMECOP line 10 is the cause of the fail.

```
static inline uint32_t gf4v_mul_2_u32(uint32_t a) {
    uint32_t bit0 = a & 0x55555555;
    uint32_t bit1 = a & 0xaaaaaaaa;
    return (bit0 << 1) ^ bit1 ^ (bit1 >> 1);
}

static inline uint32_t gf4v_mul_u32(uint32_t a, uint8_t b) {
    uint32_t bit0_b = ((uint32_t) 0) - ((uint32_t)(b & 1));
    uint32_t bit1_b = ((uint32_t) 0) - ((uint32_t)((b >> 1) & 1));
    return (a & bit0_b) ^ (bit1_b & gf4v_mul_2_u32(a));
}
```

wforcentrup

The ‘ref’, ‘ref2’ and ‘simpler’ implementations of the ‘653’, ‘761’, ‘857’, ‘953’, ‘1013’ and ‘1277’ versions failed on TIMECOP.
Flag ‘O3’ failed on all Clang versions except for ‘simpler’ where it did **not** fail for Clang ‘15.0.0’ and ‘15.0.6’:

```
==29069== Conditional jump or move depends on uninitialised value(s)
==29069== at 0x10B694:
    crypto_core_wforcesntrup953_simpler_constbranchindex (wforce.c:24)
==29069== by 0x109485: test (try.c:106)
==29069== by 0x10ADF2: main (try-anything.c:345)
==29069== Uninitialised value was created by a client request
==29069== at 0x10AC18: poison (try-anything.c:281)
==29069== by 0x109448: test (try.c:103)
==29069== by 0x10ADF2: main (try-anything.c:345)
==29069==
==29069== Conditional jump or move depends on uninitialised value(s)
==29069== at 0x10B734:
    crypto_core_wforcesntrup953_simpler_constbranchindex (wforce.c:25)
==29069== by 0x109485: test (try.c:106)
==29069== by 0x10ADF2: main (try-anything.c:345)
==29069== Uninitialised value was created by a client request
==29069== at 0x10AC18: poison (try-anything.c:281)
==29069== by 0x109448: test (try.c:103)
==29069== by 0x10ADF2: main (try-anything.c:345)
```

The line from the *wforce.c* file mentioned looks like follows.
TIMECOP mentions line 10 and 11 are the cause of the fail.

```
/* out = in if bottom bits of in have weight w */
/* otherwise out = (1,1,...,1,0,0,...,0) */
int crypto_core(unsigned char *outbytes,const unsigned char *inbytes,const
    unsigned char *kbytes,const unsigned char *cbytes)
{
  small *out = (void *) outbytes;
  const small *in = (const void *) inbytes;
  int i,mask;

  mask = Weightw_mask(in); /* 0 if weight w, else -1 */
  for (i = 0;i < w;++i) out[i] = ((in[i]^1)&~mask)^1;
  for (i = w;i < p;++i) out[i] = in[i]&~mask;
  return 0;
}
```

curve25519

The ‘ref’ implementation failed TIMECOP.
Flag ‘Os’ failed on all Clang versions:

```
==20606== Use of uninitialised value of size 8
==20606== at 0x10B9FC: freeze (smult.c:59)
==20606== by 0x10B9FC: crypto_scalarmult_curve25519_ref_constbranchindex
     (???:264)
==20606== by 0x1094BE: test (try.c:126)
==20606== by 0x10A969: main (try-anything.c:345)
```

Flag ‘O3’ failed on 17.0.2, 17.0.6 and 18.1.1:

```
==19732== Conditional jump or move depends on uninitialised value(s)
==19732== at 0x111035: freeze (smult.c:59)
==19732== by 0x111035: crypto_scalarmult_curve25519_ref_constbranchindex
     (???:264)
==19732== by 0x1094F8: test (try.c:126)
==19732== by 0x10AFFE: main (try-anything.c:345)
```

The ‘freeze’ function mentioned in the output looks like follows.
According to TIMECOP line 10 is the cause of the fail.

```
static void freeze(unsigned int a[32])
{
  unsigned int aorig[32];
  unsigned int j;
  unsigned int negative;

  for (j = 0;j < 32;++j) aorig[j] = a[j];
  add(a,a,minusp);
  negative = -((a[31] >> 7) & 1);
  for (j = 0;j < 32;++j) a[j] ^= negative & (aorig[j] ^ a[j]);
}
```

kummer

The ‘ref’ and ‘ref5u’ implementation failed TIMECOP.
Flag ‘Os’ failed on Clang versions 16.0.0, 16.0.4, 17.0.2, 17.0.6 and 18.1.1:

```
==28739== Conditional jump or move depends on uninitialised value(s)
==28739== at 0x10B5CC: cswap4x (smult.c:12)
==28739== by 0x10B5CC: crypto_scalarmult_kummer_ref5u_constbranchindex
    (???:115)
==28739== by 0x1094BE: test (try.c:126)
==28739== by 0x10AA59: main (try-anything.c:345)
==28739== Uninitialised value was created by a client request
==28739== at 0x10A911: poison (try-anything.c:281)
==28739== by 0x1094AB: test (try.c:125)
==28739== by 0x10AA59: main (try-anything.c:345)
```

The ‘cswap4x’ function mentioned in the output looks like follows.
According to TIMECOP line 8 causes the fail.

```
1  static void cswap4x(gfe *x, gfe *y, int b)
2  {
3    crypto_uint32 db = -b;
4    crypto_int32 t;
5    int i,j;
6    for(i=0;i<4;i++)
7      for(j=0;j<5;j++) {
8        t = x[i].v[j] ^ y[i].v[j];
9        t &= db;
10       x[i].v[j] ^= t;
11       y[i].v[j] ^= t;
12     }
13 }
```

Checking the assembly

To further confirm our suspicions we can try to compare the generated assembly with and without optimization. Here we can use a website called Godbolt to help view the assembly. Some of the code need to be merged and modified a bit to load it into Godbolt correctly, see the GitHub in the appendix for the code used.

invhrss701

Below is a fragment of assembly which represents the CMOV function from Invhrss compiled with O3.

```
testb $2, %r14b
vmovdqa %ymm7, 64(%rsp) # 32-byte Spill
je .LBB0_30
leaq 96(%rsp), %rdi
leaq 1504(%rsp), %rsi
movl $1402, %edx # imm = 0x57A
```

```
7 vzeroupper
8 callq memcpy@PLT
```

This conditional jump and subsequent call to memcpy when compared to the CMOV assembly code generated without optimization (see appendix 1) could explain the TIMECOP failure.

rainbowcalsecret

For rainbowcalsecret I did not manage to find the failing segment in the optimized assembly as the assembly was too complicated for me to understand.

wforcentrup

wforcentrup had two lines that failed TIMECOP. Looking at a fragment from the optimized assembly (see appendix 2 and 3), we see that a compare on line 1 and conditional jump on line 2 potentially skips a bunch of instructions. The jump instruction at the end of the assembly also seems to skip a for loop.
Later in the assembly line 1 and line 2 occur again, which as expected again skips a for loop. Matching the two lines that TIMECOP marked as the cause of failure due to conditional jumps.

curve25519

curve25519 had one line that failed TIMECOP. The 'freeze' function that failed comes after the 'mult' function. At the end of the mult function the function 'squeeze' is called. This call can be seen on line 2 of the optimized assembly (see appendix 5), so we know this is the correct spot in the assembly. In the optimized assembly we can see a conditional jump on line 18 which is most likely the cause of the TIMECOP failure.

kummer

kummer had one line that failed TIMECOP. Although I managed to compile the source code with Godbolt, I did not figure out why it would have failed on TIMECOP. For reference both the optimized and non optimized assembly has been included (see appendix 6 and 7).

Propagation

Now that we have shown that these subroutine implementations are affected by the optimization behaviour of the compiler, we can look into where these are used.

Using the fragments that failed in TIMECOP we can make some regex strings with which we can search the SUPERCOP suite for other implementations that might be vulnerable without needing to test the entire suite.

invhrss701 regex

```
for *\([^=]*=[^;]*;[^<>]*[<|>][^;]*;[^)]*\) *\r?\n? *[^\[]*\[[^\]]*\]
    *\^= *[^&]*& *\([^\[]*\[[^\]]*\] *\^ *[^\[]*\[[^\]]*\] *\)
```

wforcesntrup regex

```
for *\([^=]*=[^;]*;[^<>]*[<|>][^;]*;[^)]*\)[^=]*\[[^\]]*\] *= *\( *\(
    *[^)]*\) *& *[~|!] *[^\)]*\)\^[^;];
```

Most of the matches do not share the same vulnerabilities as the original subroutine implementations we tested, but there was one of interest.

sntrup761

sntrup761, also known as Streamlined NTRU Prime, is based on wforcesntrup and shares some of its code. Using Clang 17.0.6 with O3 it produces the following TIMECOP failure.

```
==15973== Conditional jump or move depends on uninitialised value(s)
==15973== at 0x116AD1: Decrypt (kem.c:405)
==15973== by 0x116AD1: ZDecrypt (???:713)
==15973== by 0x116AD1: Decap (???:874)
==15973== by 0x116AD1: crypto_kem_sntrup761_ref_constbranchindex_dec
    (???:899)
==15973== by 0x10B33D: main (try-anything.c:357)
==15973== Uninitialised value was created by a client request
==15973== at 0x10B218: poison (try-anything.c:281)
==15973== by 0x10B218: randombytes_callback (try-anything.c:312)
==15973== by 0x10D75F: urandom32 (kem.c:330)
==15973== by 0x10D75F: Short_random (???:343)
==15973== by 0x10D75F: KeyGen (???:375)
==15973== by 0x10D75F: ZKeyGen (???:688)
==15973== by 0x10D75F: KEM_KeyGen (???:825)
==15973== by 0x10D75F: crypto_kem_sntrup761_ref_constbranchindex_keypair
    (???:887)
==15973== by 0x109423: predoit (try.c:104)
==15973== by 0x10B303: main (try-anything.c:351)
```

sntrup761 is a Key Encapsulation Mechanism (KEM) that facilitates key establishment between clients. NTRU Prime which sntrup761 is a part of is a lattice-based cryptography system and is currently a candidate in the third-round of NIST's Post-Quantum Cryptography Standardization Project. Its post-quantum nature makes it interesting to integrate into existing code

bases. OpenSSH is an example of a project that has already integrated it and even currently uses it as the default in a hybrid scheme with x25519 (the previous default). The 9.0 release notes (see appendix for a link) mention this choice was made with the development of quantum computers in mind, as a way to prevent 'capture now, decrypt later' attacks.

Inspecting OpenSSH

The OpenSSH project contains a file called 'sntrup761.c'. This is a copy of the 'ref' implementation of sntrup761, as it is in SUPERCOP, with some minor adjustments and some currently unused functions.
The NTRU Prime software page (see appendix for link) mentions that the 'ref' and 'factored' implementations in SUPERCOP are the official reference and optimized implementations, so it makes sense that the version included in OpenSSH was taken from SUPERCOP.
There is also a script 'sntrup761.sh' that aids in integrating the SUPERCOP version of sntrup761 into OpenSSH.
Although compiling OpenSSH with a newer version of Clang went smoothly, I was unsuccessful in actually testing it with TIMECOP to see if it gave constant-time violations. It either produced no information or gave an unreasonably large number of constant-time violations.
Instead we can test sntrup761 within SUPERCOP but with the compilation flags that are used in OpenSSH to give an indication of whether there is a risk at all.
sntrup761 is normally compiled with the following flags in OpenSSH.

```
-g -O2 -pipe -Wunknown-warning-option -Qunused-arguments -Wall -Wextra -
    Wpointer-arith -Wuninitialized -Wsign-compare -Wformat-security -
    Wsizeof-pointer-memaccess -Wno-pointer-sign -Wno-unused-parameter -Wno-
    unused-result -Wmisleading-indentation -Wbitwise-instead-of-logical -
    fno-strict-aliasing -D_FORTIFY_SOURCE=2 -ftrapv -ftrivial-auto-var-init
    =zero -mretpoline -fno-builtin-memset -fstack-protector-strong -fPIE
```

However, SUPERCOP did not compile correctly with the '-ftrapv' flag set. We also need to remove the warning flags otherwise SUPERCOP would not work due to the filename length limit on Linux.
Finally we add '-march=native' and '-lgmp' to get SUPERCOP to compile everything without errors.
So the final flags used for this test were:

```
-march=native -g -O2 -pipe -Qunused-arguments -fno-strict-aliasing -
    D_FORTIFY_SOURCE=2 -ftrivial-auto-var-init=zero -mretpoline -fno-
    builtin-memset -fstack-protector-strong -fPIE -lgmp
```

Even with the lowered optimization level used in OpenSSH the 'ref' and 'compact' implementations still failed, the 'avx' and 'factored' implementa-

tions did not.
The ‘compact’ TIMECOP fail looks like this.

```
==280055== Conditional jump or move depends on uninitialised value(s)
==280055== at 0x126504: Decrypt (kem.c:332)
==280055== by 0x126504: ZDecrypt (???:409)
==280055== by 0x126504: crypto_kem_sntrup761_compact_constbranchindex_dec
    (???:468)
==280055== by 0x109791: test (try.c:158)
==280055== by 0x10B69F: main (try-anything.c:345)
==280055== Uninitialised value was created by a stack allocation
==280055== at 0x11F02A: crypto_kem_sntrup761_compact_constbranchindex_dec (
    kem.c:461)
==280055==
==280055== Conditional jump or move depends on uninitialised value(s)
==280055== at 0x126738: Decrypt (kem.c:333)
==280055== by 0x126738: ZDecrypt (???:409)
==280055== by 0x126738: crypto_kem_sntrup761_compact_constbranchindex_dec
    (???:468)
==280055== by 0x109791: test (try.c:158)
==280055== by 0x10B69F: main (try-anything.c:345)
==280055== Uninitialised value was created by a stack allocation
==280055== at 0x11F02A: crypto_kem_sntrup761_compact_constbranchindex_dec (
    kem.c:461)
```

And the ‘ref’ implementation which is used in OpenSSH failed like this.

```
==280426== Conditional jump or move depends on uninitialised value(s)
==280426== at 0x117A21: Decrypt (kem.c:405)
==280426== by 0x117A21: ZDecrypt (???:713)
==280426== by 0x117A21: Decap (???:874)
==280426== by 0x117A21: crypto_kem_sntrup761_ref_constbranchindex_dec
    (???:899)
==280426== by 0x109791: test (try.c:158)
==280426== by 0x10B69F: main (try-anything.c:345)
==280426== Uninitialised value was created by a stack allocation
==280426== at 0x11516A: crypto_kem_sntrup761_ref_constbranchindex_dec (kem.c
    :898)
==280426==
==280426== Conditional jump or move depends on uninitialised value(s)
==280426== at 0x117AA8: Decrypt (kem.c:406)
==280426== by 0x117AA8: ZDecrypt (???:713)
==280426== by 0x117AA8: Decap (???:874)
==280426== by 0x117AA8: crypto_kem_sntrup761_ref_constbranchindex_dec
    (???:899)
==280426== by 0x109791: test (try.c:158)
==280426== by 0x10B69F: main (try-anything.c:345)
==280426== Uninitialised value was created by a stack allocation
==280426== at 0x11516A: crypto_kem_sntrup761_ref_constbranchindex_dec (kem.c
    :898)
```

The two lines that failed still match with what we found originally in sntrup761 and wforcesntrup761.

Discussion

The code snippets that failed TIMECOP and caused the undesirable behaviour in the compiler do not look particularly odd, they seem to follow the constant time programming paradigm as expected. By looking at the results of Table 4.2, where all the tests that were compiled without optimization passed, we can also conclude that the source code was written in a constant-time way. Compiling without optimization should have yielded at least some TIMECOP fails if the code was not constant-time, but that does not appear to be the case.

In the case of both the 'O3' flag and the 'Os' flag, it seems that the optimizations did turn constant-time source code into variable-time binaries. Potentially introducing timing attacks as a weakness.
This confirms that standard compiler optimizations like in Clang have a realistic chance of causing constant-time source code to be turned into variable-time binaries.

Looking further into the results in Table 4.1, it seems that the more recent the version of Clang, the more TIMECOP failures there are.
A speculative explanation for this effect would be that over time more 'intelligent' optimizations were added to the compilers. Where older versions of the compiler might take the constant-time source code at face value, later iterations of the compiler perhaps mistake the round-about methods of constant-time programming as inefficiency, instead optimizing away the constant-time programming techniques and reintroducing branching, unsafe index accesses and variable-time instructions.

As we have shown with NTRU Prime's `sntrup761` and OpenSSH, this behaviour does pose a risk. Basic building blocks are worked into schemes which in turn become part of larger projects and systems. Whether or not a system is safe then hinges on whether the developers and maintainers in this chain continue to verify that their compiled programs are secure. Even if no changes to the code have been made, as simply using a newer compiler version could reveal weaknesses that were not there before. This highlights that extra care needs to be taken when integrating and compiling cryptographic implementations written in C. A project's default configuration might already be enough to open up avenues of attack when using newer compiler versions.

Chapter 5

Conclusions

We've shown that the optimizations performed by Clang do indeed have the chance of turning constant-time source code into variable-time binaries and that with newer versions of Clang (17+) the chance of this happening is higher.
We also demonstrated that the reference implementation of `sntrup761` from the NTRU Prime project, which has been integrated into OpenSSH, could be vulnerable to this behaviour when compiling with Clang 17.0.6. Fortunately, there is a wide ecosystem of tools available to developers to detect constant-time violations and special languages that prevent this situation entirely. Nevertheless, thepossibility of optimizations weakening security is something that should be kept in mind, especially when using C in cryptography contexts.

Future work

Using the method described in this paper more cryptographic implementations that are not a part of the 'used' subroutines section in SUPERCOP could be tested. Mainly so there are more examples available of this unwanted optimization behaviour.
The tests could also be repeated with other compilers and a wider range of compiler versions to confirm if this pattern of newer compilers producing more constant-time failures after optimization persists.

More research can be done towards determining the exact cause of this undesirable behaviour. Specifically which optimizations, that are a part of the optimization levels, cause the behaviour.

Finally it would be interesting to verify whether OpenSSH is truly vulnerable to a timing attack when compiled with a newer Clang version (17+), using OpenSSH's default optimization level of O2.

Bibliography

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, 2017.

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 965–982. IEEE, 2020.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Vincent Laporte, and Tiago Oliveira. Certified compilation for cryptography: Extended x86 instructions and constant-time verification. In *Progress in Cryptology–INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 13–16, 2020, Proceedings 21*, pages 107–127. Springer, 2020.

[4] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[5] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Structured leakage and applications to cryptographic constant-time and cost. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 462–476, 2021.

[6] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[7] Daniel J Bernstein. Cache-timing attacks on aes. 2005.

[8] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K Rustan M Leino, Jacob R Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying {High-Performance} cryptographic assembly code. In *26th USENIX security symposium (USENIX security 17)*, pages 917–934, 2017.

[9] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[10] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 913–926, 2020.

[11] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 666–680. IEEE, 2022.

[12] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.

[13] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1021–1038. IEEE, 2020.

[14] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/rel: symbolic binary analyzer for security with applications to constant-time and secret-erasure. *ACM Transactions on Privacy and Security*, 26(2):1–42, 2023.

[15] Craig Disselkoen, Sunjay Cauligi, Dean Tullsen, and Deian Stefan. Finding and eliminating timing side-channels in crypto code with pitchfork. In *TECHCON*, 2020.

[16] Marcel Fourné, Daniel De Almeida Braga, Jan Jancar, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "these results must be false": A usability evaluation of constant-time analysis tools.

[17] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar.

"they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 632–649. IEEE, 2022.

[18] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting curve25519-donna built with msvc 2015. In *Cryptology and Network Security: 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings 15*, pages 573–582. Springer, 2016.

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[20] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[21] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure dsa signing exponentiations really are constant-time. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1650, 2016.

[22] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1697–1702. IEEE, 2017.

[23] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[24] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[25] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7:99–112, 2017.

[26] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. Hacl*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1806, 2017.

Appendix A

Appendix

The raw test output data, Python parsing scripts and other data related can be viewed at:
`https://github.com/vdankbaar/Bachelor-Thesis`

The TIMECOP tool can be found at:
`https://www.post-apocalyptic-crypto.org/timecop/`.

The SUPERCOP tool can be found at:
`https://bench.cr.yp.to/supercop.html`.

Some Godbolt instances:
invhrss701: `https://godbolt.org/z/3ce61qxK9`
wforcentrup: `https://godbolt.org/z/GY8o7EnsP`
curve25519: `https://godbolt.org/z/fEvTPo6Gv`
kummer: `https://godbolt.org/z/avzrM15sr`
If the links are down, the source files I used can be found in the thesis GitHub repo.

The OpenSSH repository can be found at:
`https://github.com/openssh/openssh-portable`

NTRU Prime page:
`https://ntruprime.cr.yp.to/software.html`

OpenSSH 9.0 Release notes:
`https://www.openssh.com/txt/release-9.0`

1. invhrss701: CMOV (no optimization)

```
cmov: # @cmov
   movb %cl, %al
   movq %rdi, -8(%rsp)
   movq %rsi, -16(%rsp)
   movq %rdx, -24(%rsp)
   movb %al, -25(%rsp)
   movzbl -25(%rsp), %ecx
   xorl %eax, %eax
   subl %ecx, %eax
   movb %al, -25(%rsp)
   movq $0, -40(%rsp)
   .LBB5_1: # =>This Inner Loop Header: Depth=1
      movq -40(%rsp), %rax
      cmpq -24(%rsp), %rax
   jae .LBB5_4
      movzbl -25(%rsp), %esi
      movq -16(%rsp), %rax
      movq -40(%rsp), %rcx
      movzbl (%rax,%rcx), %eax
      movq -8(%rsp), %rcx
      movq -40(%rsp), %rdx
      movzbl (%rcx,%rdx), %ecx
      xorl %ecx, %eax
      andl %eax, %esi
      movq -8(%rsp), %rax
      movq -40(%rsp), %rcx
      movzbl (%rax,%rcx), %edx
      xorl %esi, %edx
      movb %dl, (%rax,%rcx)
      movq -40(%rsp), %rax
      addq $1, %rax
      movq %rax, -40(%rsp)
      jmp .LBB5_1
.LBB5_4:
   retq
```

2. wforcentrup (no optimization)

```
.LBB0_1: # =>This Inner Loop Header: Depth=1
       cmp dword ptr [rsp + 4], 286
       jge .LBB0_4
       mov rax, qword ptr [rsp + 8]
       movsxd rcx, dword ptr [rsp + 4]
       movsx eax, byte ptr [rax + rcx]
       xor eax, 1
       mov ecx, dword ptr [rsp]
       xor ecx, -1
       and eax, ecx
       xor eax, 1
       mov dl, al
```

```
13        mov rax, qword ptr [rsp + 16]
14        movsxd rcx, dword ptr [rsp + 4]
15        mov byte ptr [rax + rcx], dl
16        mov eax, dword ptr [rsp + 4]
17        add eax, 1
18        mov dword ptr [rsp + 4], eax
19        jmp .LBB0_1
20 .LBB0_4:
21        mov dword ptr [rsp + 4], 286
22 .LBB0_5: # =>This Inner Loop Header: Depth=1
23        cmp dword ptr [rsp + 4], 761
24        jge .LBB0_8
25        mov rax, qword ptr [rsp + 8]
26        movsxd rcx, dword ptr [rsp + 4]
27        movsx eax, byte ptr [rax + rcx]
28        mov ecx, dword ptr [rsp]
29        xor ecx, -1
30        and eax, ecx
31        mov dl, al
32        mov rax, qword ptr [rsp + 16]
33        movsxd rcx, dword ptr [rsp + 4]
34        mov byte ptr [rax + rcx], dl
35        mov eax, dword ptr [rsp + 4]
36        add eax, 1
37        mov dword ptr [rsp + 4], eax
38        jmp .LBB0_5
39 .LBB0_8:
```

3. wforcentrup (optimization O3)

```
1         cmp eax, 286
2         jne .LBB0_18
3         mov rcx, rdi
4         sub rcx, rsi
5         cmp rcx, 63
6         jbe .LBB0_5
7         vmovups ymm0, ymmword ptr [rsi]
8         vmovups ymm1, ymmword ptr [rsi + 32]
9         vmovups ymmword ptr [rdi], ymm0
10        vmovups ymmword ptr [rdi + 32], ymm1
11        vmovups ymm0, ymmword ptr [rsi + 64]
12        vmovups ymm1, ymmword ptr [rsi + 96]
13        vmovups ymmword ptr [rdi + 64], ymm0
14        vmovups ymmword ptr [rdi + 96], ymm1
15        vmovups ymm0, ymmword ptr [rsi + 128]
16        vmovups ymm1, ymmword ptr [rsi + 160]
17        vmovups ymmword ptr [rdi + 128], ymm0
18        vmovups ymmword ptr [rdi + 160], ymm1
19        vmovdqu ymm0, ymmword ptr [rsi + 192]
20        vmovdqu ymm1, ymmword ptr [rsi + 224]
21        vmovdqu ymmword ptr [rdi + 192], ymm0
22        vmovdqu ymmword ptr [rdi + 224], ymm1
23        mov rdx, qword ptr [rsi + 256]
```

```
        mov qword ptr [rdi + 256], rdx
        mov rdx, qword ptr [rsi + 264]
        mov qword ptr [rdi + 264], rdx
        mov rdx, qword ptr [rsi + 272]
        mov qword ptr [rdi + 272], rdx
        mov edx, 280
        jmp .LBB0_6
.LBB0_18:
        vpbroadcastb ymm0, byte ptr [rip + .LCPI0_2]
                              # ymm0 = [1,1,1,...1,1]
        vmovdqu ymmword ptr [rdi + 254], ymm0
        vmovdqu ymmword ptr [rdi + 224], ymm0
        vmovdqu ymmword ptr [rdi + 192], ymm0
        vmovdqu ymmword ptr [rdi + 160], ymm0
        vmovdqu ymmword ptr [rdi + 128], ymm0
        vmovdqu ymmword ptr [rdi + 96], ymm0
        vmovdqu ymmword ptr [rdi + 64], ymm0
        vmovdqu ymmword ptr [rdi + 32], ymm0
        vmovdqu ymmword ptr [rdi], ymm0
        jmp .LBB0_19
```

4. curve25519: freeze (no optimization)

```
freeze: # @freeze
        sub rsp, 152
        mov qword ptr [rsp + 144], rdi
        mov dword ptr [rsp + 12], 0
.LBB4_1: # =>This Inner Loop Header: Depth=1
        cmp dword ptr [rsp + 12], 32
        jae .LBB4_4
        mov rax, qword ptr [rsp + 144]
        mov ecx, dword ptr [rsp + 12]
        mov ecx, dword ptr [rax + 4*rcx]
        mov eax, dword ptr [rsp + 12]
        mov dword ptr [rsp + 4*rax + 16], ecx
        mov eax, dword ptr [rsp + 12]
        add eax, 1
        mov dword ptr [rsp + 12], eax
        jmp .LBB4_1
.LBB4_4:
        mov rdi, qword ptr [rsp + 144]
        mov rsi, qword ptr [rsp + 144]
        lea rdx, [rip + minusp]
        call add
        mov rax, qword ptr [rsp + 144]
        mov ecx, dword ptr [rax + 124]
        shr ecx, 7
        and ecx, 1
        xor eax, eax
        sub eax, ecx
        mov dword ptr [rsp + 8], eax
        mov dword ptr [rsp + 12], 0
.LBB4_5: # =>This Inner Loop Header: Depth=1
```

```
        cmp dword ptr [rsp + 12], 32
        jae .LBB4_8
        mov edx, dword ptr [rsp + 8]
        mov eax, dword ptr [rsp + 12]
        mov eax, dword ptr [rsp + 4*rax + 16]
        mov rcx, qword ptr [rsp + 144]
        mov esi, dword ptr [rsp + 12]
        xor eax, dword ptr [rcx + 4*rsi]
        and edx, eax
        mov rax, qword ptr [rsp + 144]
        mov ecx, dword ptr [rsp + 12]
        xor edx, dword ptr [rax + 4*rcx]
        mov dword ptr [rax + 4*rcx], edx
        mov eax, dword ptr [rsp + 12]
        add eax, 1
        mov dword ptr [rsp + 12], eax
        jmp .LBB4_5
.LBB4_8:
        add rsp, 152
        ret
```

5. curve25519: freeze (optimization O3)

```
        vzeroupper
        call squeeze
        vmovups ymm0, ymmword ptr [rbx]
        vmovups ymm1, ymmword ptr [rbx + 32]
        vmovups ymm2, ymmword ptr [rbx + 64]
        vmovups ymm3, ymmword ptr [rbx + 96]
        lea rdx, [rip + minusp]
        mov rdi, rbx
        mov rsi, rbx
        vmovups ymmword ptr [rsp + 1200], ymm3
        vmovups ymmword ptr [rsp + 1168], ymm2
        vmovups ymmword ptr [rsp + 1136], ymm1
        vmovups ymmword ptr [rsp + 1104], ymm0
        vzeroupper
        call add
        mov eax, dword ptr [rsp + 2508]
        test al, al
        jns .LBB0_999
        vmovups ymm0, ymmword ptr [rsp + 1104]
        vmovups ymm1, ymmword ptr [rsp + 1136]
        vmovups ymm2, ymmword ptr [rsp + 1168]
        vmovups ymm3, ymmword ptr [rsp + 1200]
        vmovups ymmword ptr [rbx + 96], ymm3
        vmovups ymmword ptr [rbx + 64], ymm2
        vmovups ymmword ptr [rbx + 32], ymm1
        vmovups ymmword ptr [rbx], ymm0
.LBB0_999:
```

6. kummer: cswap4x (no optimization)

```
cswap4x: # @cswap4x
       mov qword ptr [rsp - 8], rdi
       mov qword ptr [rsp - 16], rsi
       mov dword ptr [rsp - 20], edx
       xor eax, eax
       sub eax, dword ptr [rsp - 20]
       mov dword ptr [rsp - 24], eax
       mov dword ptr [rsp - 32], 0
.LBB1_1: # =>This Loop Header: Depth=1
       cmp dword ptr [rsp - 32], 4
       jge .LBB1_8
       mov dword ptr [rsp - 36], 0
.LBB1_3: # Parent Loop BB1_1 Depth=1
       cmp dword ptr [rsp - 36], 5
       jge .LBB1_6
       mov rax, qword ptr [rsp - 8]
       movsxd rcx, dword ptr [rsp - 32]
       imul rcx, rcx, 20
       add rax, rcx
       movsxd rcx, dword ptr [rsp - 36]
       mov eax, dword ptr [rax + 4*rcx]
       mov rcx, qword ptr [rsp - 16]
       movsxd rdx, dword ptr [rsp - 32]
       imul rdx, rdx, 20
       add rcx, rdx
       movsxd rdx, dword ptr [rsp - 36]
       xor eax, dword ptr [rcx + 4*rdx]
       mov dword ptr [rsp - 28], eax
       mov eax, dword ptr [rsp - 24]
       and eax, dword ptr [rsp - 28]
       mov dword ptr [rsp - 28], eax
       mov edx, dword ptr [rsp - 28]
       mov rax, qword ptr [rsp - 8]
       movsxd rcx, dword ptr [rsp - 32]
       imul rcx, rcx, 20
       add rax, rcx
       movsxd rcx, dword ptr [rsp - 36]
       xor edx, dword ptr [rax + 4*rcx]
       mov dword ptr [rax + 4*rcx], edx
       mov edx, dword ptr [rsp - 28]
       mov rax, qword ptr [rsp - 16]
       movsxd rcx, dword ptr [rsp - 32]
       imul rcx, rcx, 20
       add rax, rcx
       movsxd rcx, dword ptr [rsp - 36]
       xor edx, dword ptr [rax + 4*rcx]
       mov dword ptr [rax + 4*rcx], edx
       mov eax, dword ptr [rsp - 36]
       add eax, 1
       mov dword ptr [rsp - 36], eax
       jmp .LBB1_3
.LBB1_6: # in Loop: Header=BB1_1 Depth=1
```

```
       jmp .LBB1_7
.LBB1_7: # in Loop: Header=BB1_1 Depth=1
       mov eax, dword ptr [rsp - 32]
       add eax, 1
       mov dword ptr [rsp - 32], eax
       jmp .LBB1_1
.LBB1_8:
       ret
```

7. kummer: cswap4x (optimization O3)

```
.LBB0_1: # =>This Loop Header: Depth=1
       mov qword ptr [rsp + 8], rcx # 8-byte Spill
.LBB0_2: # Parent Loop BB0_1 Depth=1
       mov dword ptr [rsp + 4], esi # 4-byte Spill
       mov eax, ebx
       mov rcx, qword ptr [rsp + 8] # 8-byte Reload
       mov rdx, qword ptr [rsp + 272] # 8-byte Reload
       movzx ecx, byte ptr [rdx + rcx]
       shrx ebx, ecx, esi
       and ebx, 1
       xor eax, ebx
       vmovdqu ymm0, ymmword ptr [rsp + 176]
       neg eax
       vpbroadcastd ymm1, eax
       vmovdqu ymm2, ymmword ptr [rsp + 96]
       vmovdqu ymm3, ymmword ptr [rsp + 128]
       vmovdqa ymm4, ymm1
       vpternlogd ymm4, ymm0, ymm2, 96
       vpxor ymm2, ymm4, ymm2
       vmovdqu ymmword ptr [rsp + 96], ymm2
       vmovdqu ymm2, ymmword ptr [rsp + 208]
       vmovdqa ymm5, ymm1
       vpternlogd ymm5, ymm2, ymm3, 96
       vperm2i128 ymm6, ymm4, ymm5, 33 # ymm6 = ymm4[2,3],ymm5[0,1]
       vperm2i128 ymm0, ymm0, ymm2, 33 # ymm0 = ymm0[2,3],ymm2[0,1]
       vpxor ymm0, ymm6, ymm0
       vmovdqu ymmword ptr [rsp + 192], ymm0
       vpxor ymm0, ymm5, ymm3
       vmovdqu ymmword ptr [rsp + 128], ymm0
       vmovdqa xmm0, xmmword ptr [rsp + 160]
       vmovdqa xmm2, xmmword ptr [rsp + 240]
       vpternlogd xmm1, xmm2, xmm0, 96
       vinserti128 ymm3, ymm1, xmm4, 1
       vinserti128 ymm4, ymm0, xmmword ptr [rsp + 176], 1
       vpblendd ymm0, ymm4, ymm0, 15 # ymm0 = ymm0[0,1,2,3],ymm4[4,5,6,7]
       vpxor ymm0, ymm3, ymm0
       vmovdqu ymmword ptr [rsp + 160], ymm0
       vextracti128 xmm0, ymm5, 1
       vinserti128 ymm0, ymm0, xmm1, 1
       vmovdqu ymm1, ymmword ptr [rsp + 224]
       vinserti128 ymm1, ymm1, xmm2, 1
       vpxor ymm0, ymm0, ymm1
```

```
43          vmovdqu ymmword ptr [rsp + 224], ymm0
```