

A Constant-Time Analysis Framework for Post-Quantum Cryptography

Pablo Gutiérrez-Félix

Workshop on Secure Protocol Implementations in the Quantum Era

EUROS&P 2026, Lisbon, Portugal



pablogf@uma.es

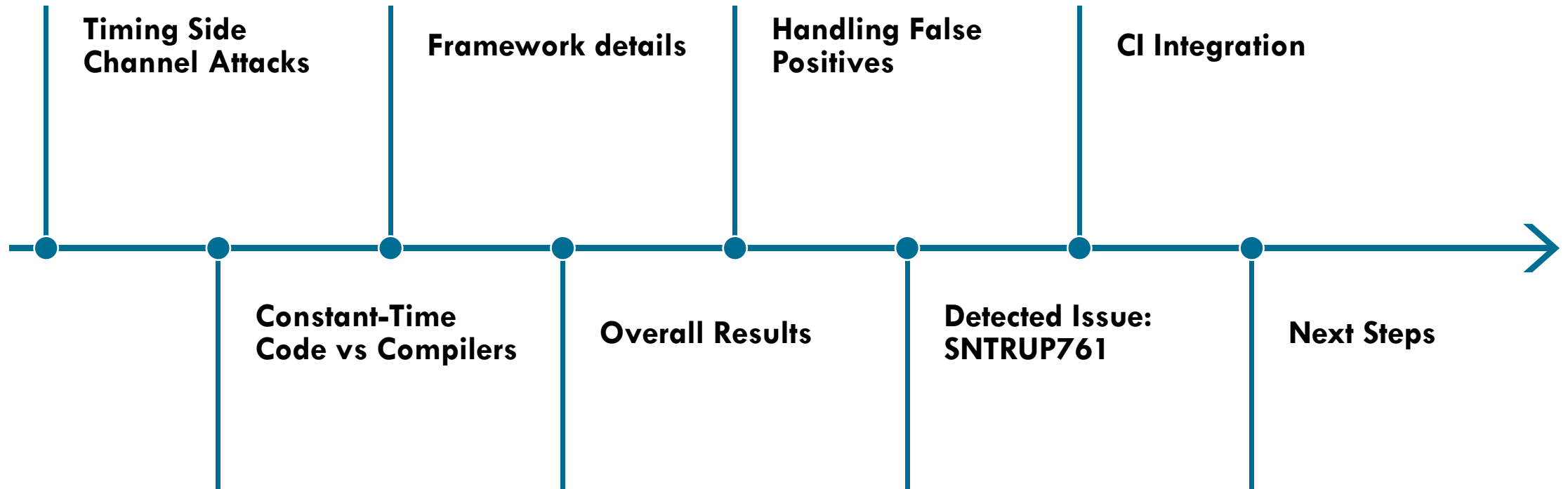


linkedin.com/in/pablogutierrezfelix



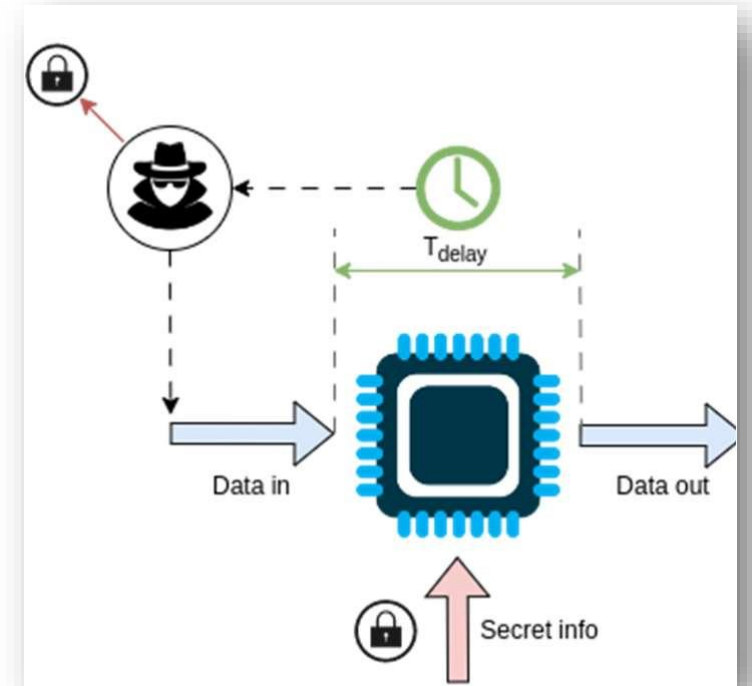
github.com/pablo-gf

Agenda



Timing Side Channel Attacks

- Specific family of SCAs: Non-invasive, **measures execution time**
- Why do timing leaks originate?
 - **Secret-dependent control flow**
 - Conditional branching: RSA/DH's square and multiply
 - Table lookups: AES
 - **Compiler optimizations**
 - Microarchitectural behavior



- Solution: Converting conditional branches to **constant-time code**

Constant-Time Code: Clangover

- Kyber message decompression: Contains a conditional branch

Naive implementation



```
void poly_frommsg(poly *r, const uint8_t msg[KYBER_INDCPA_MSGBYTES])
{
    unsigned int i,j;

    #if (KYBER_INDCPA_MSGBYTES != KYBER_N/8)
    #error "KYBER_INDCPA_MSGBYTES must be equal to KYBER_N/8 bytes!"
    #endif

    for(i=0;i<KYBER_N/8;i++) {
        for(j=0;j<8;j++) {
            if (msg[i] & (i << j))
                r->coeffs[8*i+j] = ((KYBER_Q+1)/2);
            else
                r->coeffs[8*i+j] = 0;
        }
    }
}
```

Original implementation



```
void poly_frommsg(poly *r, const uint8_t msg[KYBER_INDCPA_MSGBYTES])
{
    unsigned int i,j;
    int16_t mask;

    #if (KYBER_INDCPA_MSGBYTES != KYBER_N/8)
    #error "KYBER_INDCPA_MSGBYTES must be equal to KYBER_N/8 bytes!"
    #endif

    for(i=0;i<KYBER_N/8;i++) {
        for(j=0;j<8;j++) {
            mask = -(int16_t)((msg[i] >> j)&1);
            r->coeffs[8*i+j] = mask & ((KYBER_Q+1)/2);
        }
    }
}
```

<https://github.com/antoonpurnal/clangover>

Compiler optimizations: Clangover

- Clang ≥ 15 on x86-64 with certain flags optimizes to conditional branch
 - Key recovery attack in 5-10 min on ML-KEM-512

Without

```
1 poly_frommsg:
2     xor     eax, eax
3 .LBB0_1:
4     xor     eax, eax
5 .LBB0_2:
6     movzx   eax, byte_0
7     shr     eax, 8
8     and     eax, 0xFF
9     neg     eax
10    and     eax, 0xFF
11    mov     eax, eax
```

Constant-Time Code: The Pessimist Case

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

6 March, 2025

Abstract. This note discusses the problem of writing cryptographic implementations in software, free of timing-based side-channels, and many ways in which that endeavour can fail in practice. It is a pessimist view: it highlights why such failures are expected to become more common, and how constant-time coding is, or will soon become, infeasible in all generality.

flags



ptr [rsi + r8]

<https://github.com/antoonpurnal/clangover>

Selected Tools

- Valgrind
 - MemCheck helps identify non-constant time implementations
 - Operates directly on binaries compiled for **specific architectures**
 - Kyberslash patch for **variable-latency errors in division**
- MemSan
 - Follows the same functionality enabled by Valgrind's MemCheck
 - Operates at the LLVM-IR level: **Multi-platform**
 - Native to clang compiler



IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2025, No. 2, pp. 209–234. DOI:10.46586/tches.v2025.i2.209-234

KyberSlash: Exploiting secret-dependent division timings in Kyber implementations

Daniel J. Bernstein^{1,2}, Karthikeyan Bhargavan^{3,4}, Shivam Bhasin^{5,7}, Anupam Chattopadhyay^{6,7}, Tee Kiah Chia⁷, Matthias J. Kannwischer⁸, Franziskus Kiefer⁴, Thales B. Paiva^{9,10,11}, Prasanna Ravi^{6,7} and Goutam Tamvada⁴

Adam Langley. ctgrind. <https://github.com/agl/ctgrind>
<https://valgrind.org/info/platforms.html>
<https://kyberslash.cr.yp.to/index.html>
<https://clang.llvm.org/docs/MemorySanitizer.html#supported-platforms>

Poisoning Secrets

- Warns if undefined variable is used in a table lookup/conditional branch
 - This mirrors the task we are required to do for constant-time analysis!
- Make secrets undefined, the output will tell if there are constant-time issues

VALGRIND_MAKE_MEM_UNDEFINED()

```
==3414==WARNING: MemorySanitizer: use-of-uninitialized-value
#0 0x7ffff7f51522 in secure_l32_mask /home/pablogf/liboqs/src/kem/bike/additional_r4/utilities.h:122:3
#1 0x7ffff7f51522 in OQS_KEM_bike_l1_decaps /home/pablogf/liboqs/src/kem/bike/additional_r4/kem.c:259:19
#2 0x55555561fb67 in kem_test_correctness /home/pablogf/liboqs/tests/test_kem.c:301:7
#3 0x55555561f48d in test_wrapper /home/pablogf/liboqs/tests/test_kem.c:388:11
#4 0x7ffff7a9caa3 in start_thread nptl/pthread_create.c:447:8
#5 0x7ffff7b29c6b in clone3 misc/../sysdeps/unix/sysv/linux/x86_64/clone3.S:78

SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/pablogf/liboqs/src/kem/bike/additional_r4/utilities.h:122:3 in secure_l32_mask
```

__msan_poison()

```
====
==6037== Thread 2:
==6037== Conditional jump or move depends on uninitialised value(s)
==6037== at 0x404DE82: mlk_rej_uniform_scalar (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/sampling.c:62)
==6037== by 0x404DA20: mlk_rej_uniform (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/sampling.c:137)
==6037== by 0x404D7D9: PQCP_MLKEM_NATIVE_MLKEMS12_C_poly_rej_uniform_x4 (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/sampling.c:170)
==6037== by 0x414B4D8: PQCP_MLKEM_NATIVE_MLKEMS12_C_gen_matrix (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/indcpa.c:246)
==6037== by 0x414B6D0: PQCP_MLKEM_NATIVE_MLKEMS12_C_indcpa_keypair_derand (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/indcpa.c:360)
==6037== by 0x404C274: PQCP_MLKEM_NATIVE_MLKEMS12_C_keypair_derand (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/kem.c:215)
==6037== by 0x404C36C: PQCP_MLKEM_NATIVE_MLKEMS12_C_keypair (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/kem.c:248)
==6037== by 0x404C16C: OQS_KEM_ml_kem_512_keypair (src/kem/ml_kem/mlkem-native_ml-kem-512_ref/mlkem/src/kem.c:134)
==6037== by 0x40084EC: OQS_KEM_keypair (src/kem/kem.c:559)
==6037== by 0x40049D3: kem_test_correctness (tests/test_kem.c:221)
==6037== by 0x40042C1: test_wrapper (tests/test_kem.c:378)
==6037== by 0x4030AA3: start_thread (pthread_create.c:447)
==6037==
=====
```

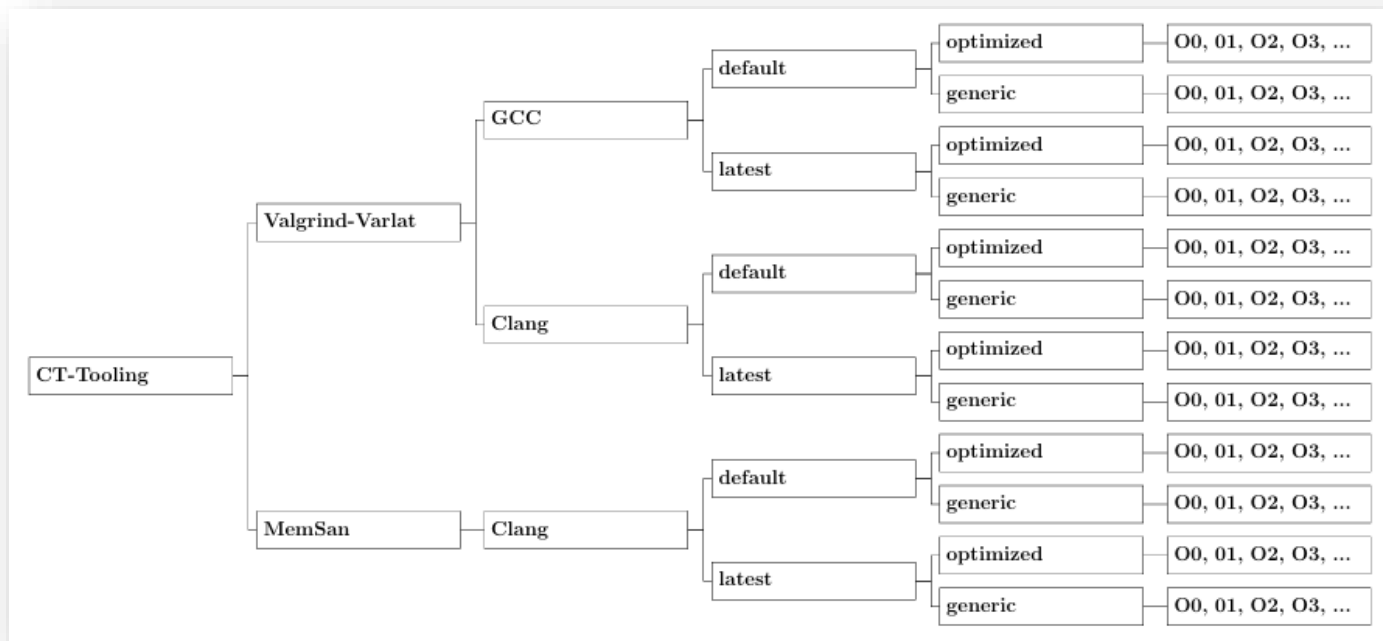
LIBOQS Integration

- Why liboqs?
 - Widely used library for post-quantum algorithms
 - Continually updating and including new implementations
- How to do it?
 - Include a specific test for CT that calls the poisoning of **randombytes()**
 - “Unpoison” public keys, and `verify()` functions for signatures
 - No danger: They are publicly available
 - Store all unique warnings output by the chosen tooling into algorithm logs
 - Log files include the function, file, line of code, and execution trace



Framework Design

- Open-source → end user compiles source code
- There exist many compilation parameters: **Compiler, version, build, flags, ...**



Optimization Flag	Purpose
-O0	No optimization
-O1	Basic optimization
-O2 -fno-tree-vectorize (-fno-vectorize for clang)	Typical optimization minus automatic vectorization
-O2	Typical optimization
-Os	Same optimization level as -O2 but favors smaller code size over speed
-O3 -fno-tree-vectorize (-fno-vectorize for clang)	More aggressive optimizations (loop unrolling, inlining) but no vectorization
-O3	Very aggressive optimizations including vectorization
-Ofast	Maximum optimization, even if breaking standards

Overall Results

Valgrind-Varlat/Clang

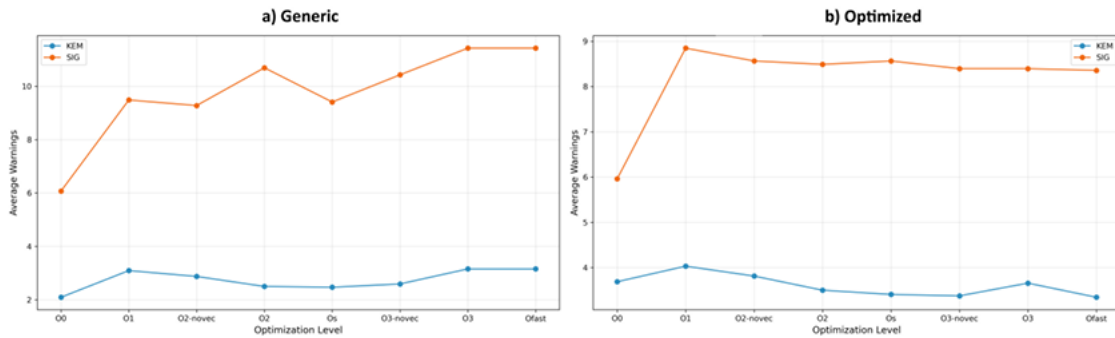


Figure 3: Valgrind-Varlat average KEM and SIG warnings per optimization level compiled with clang-18. The left graph shows results from compiling with generic build, and the right graph with the optimized build.

Valgrind-Varlat/GCC

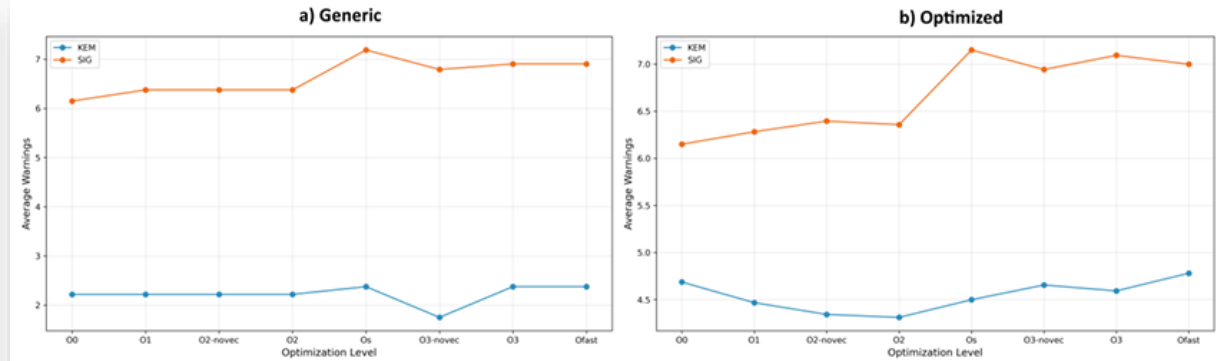


Figure 7: Valgrind-Varlat average KEM and SIG warnings per optimization level compiled with gcc-13. The left graph shows the results from compiling with the generic build, and the right graph shows the optimized build.

MemSan/Clang

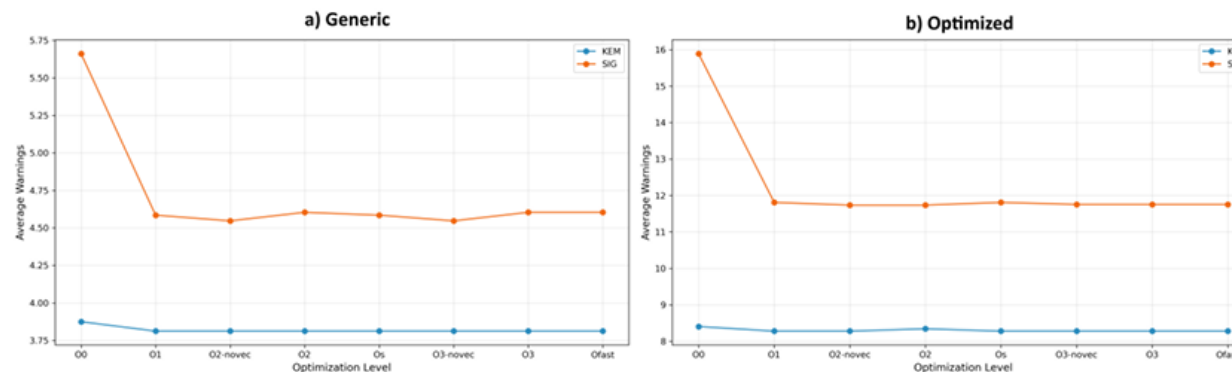


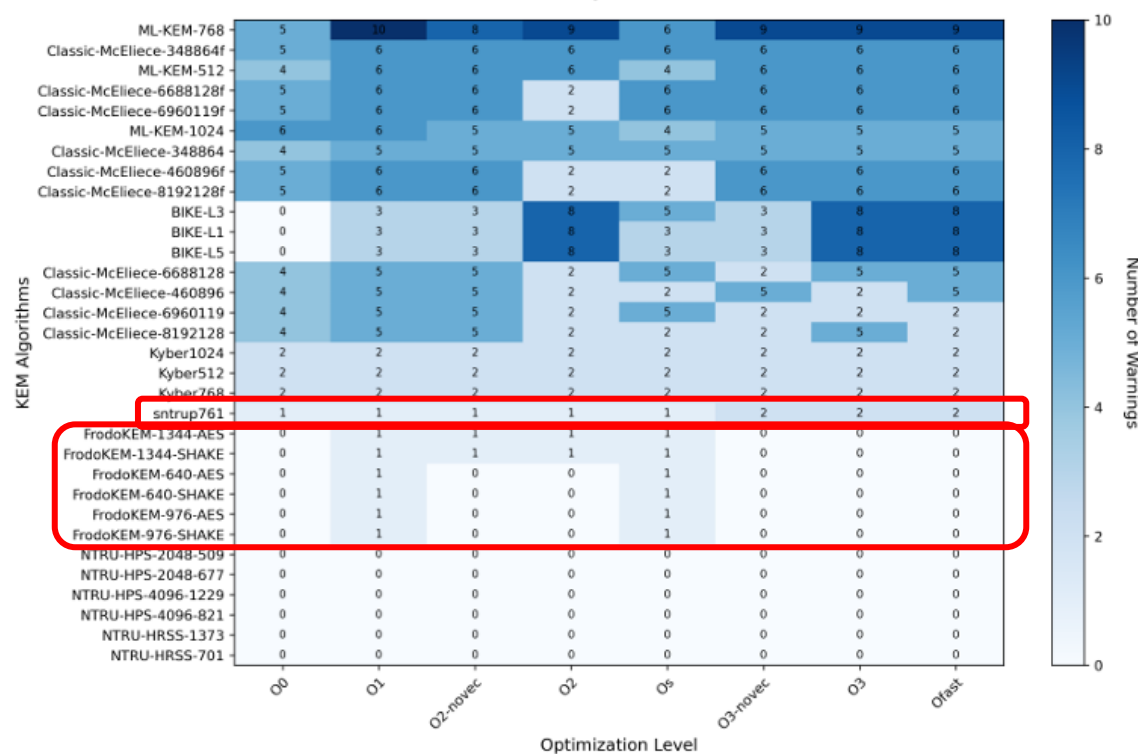
Figure 11: MemSan average KEM and SIG warnings per optimization level compiled with clang-18. The left graph shows the results from compiling with the generic build, and the right graph shows the optimized build.

Example - Valgrind-Varlat/clang (KEMs)

SNTRUP761: int16_nonzero_mask()

- clang-20 clang-13 clang-19 / generic / -O3 -Ofast
- Addressed in SUPERCOP and OpenSSH: liboqs is outdated

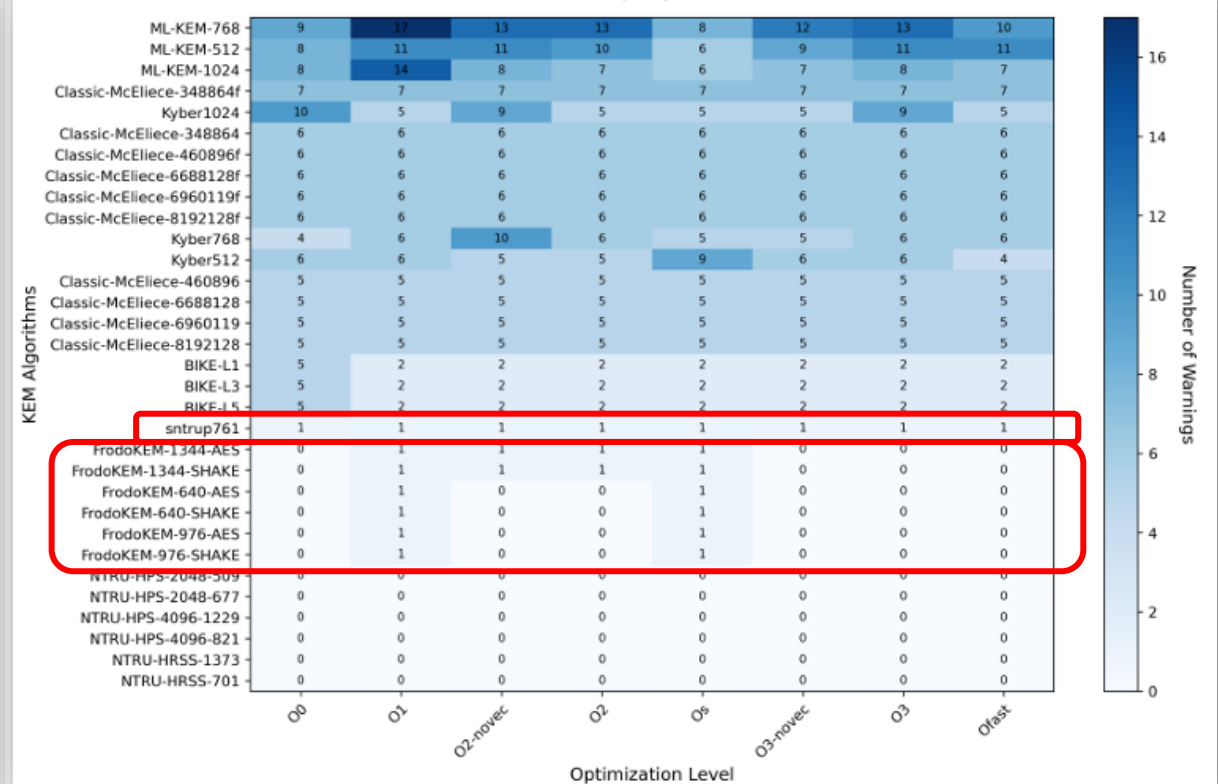
a) Generic



FrodoKEM: ct_select()

- clang-18 clang-20 / generic and optimized / -O1, -O2, -Os

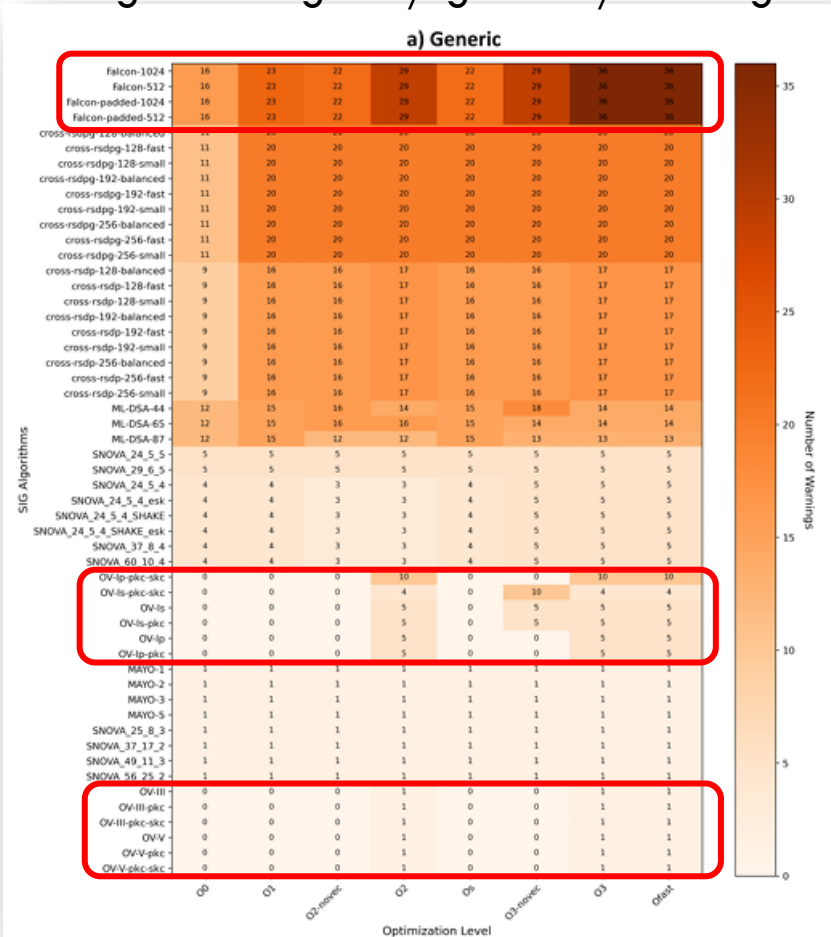
b) Optimized



Example: Valgrind-Varlat/clang (SIGs)

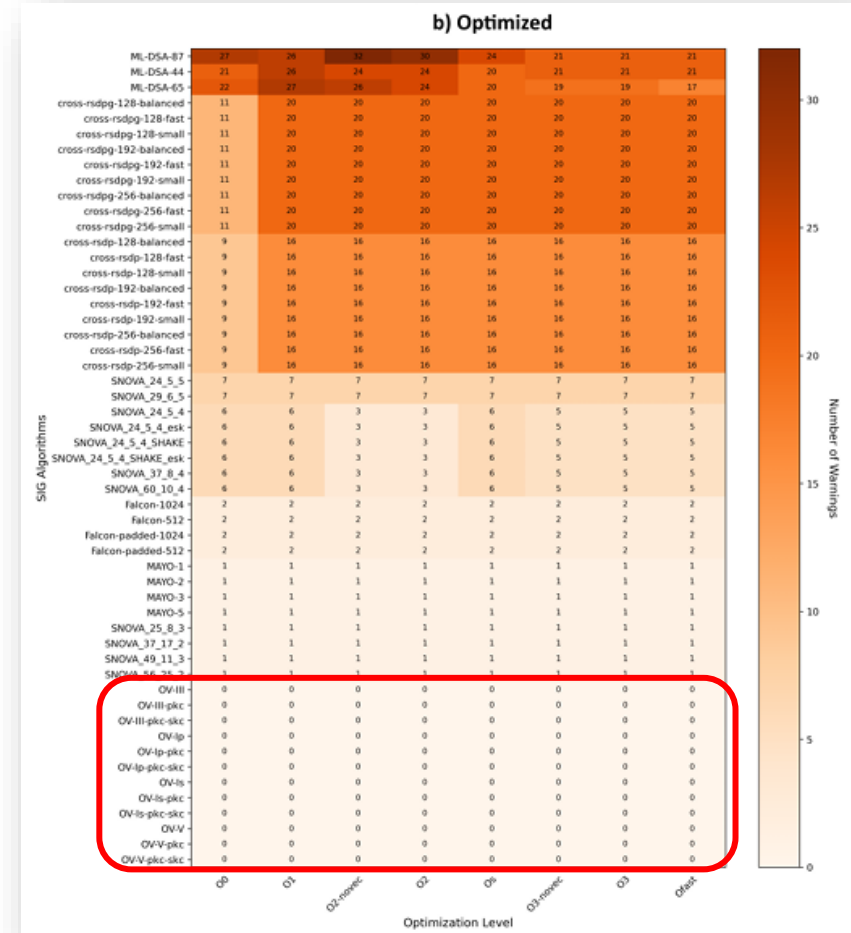
OV: gf16v_mul_u32, batch_quad_trimat_eval_gf16, gf16mat_gauss_elim_row_echeleon

- clang-18 clang-20 / generic / all flags



Falcon: zint_sub, solve_NTRU, sint_norm_zero

- clang-18 and gcc / generic / -O3 -Ofast



Handling False Positives

- **Problem:** The tooling sometimes outputs false positives
 - Example: During public parameter rejection sampling in ML-KEM and ML-DSA
- **Solution:** Passing suppression files to the tools during testing

Valgrind-Varlat

```
{
  Rejection sampling for small (f,g) coefficients
  Memcheck:Cond
  src:keygen.c:4153 # fun:PQCLEAN_FALCON*_*_keygen
  fun:PQCLEAN_FALCON*_AVX2_crypto_sign_keypair
}
{
  Rejection sampling for small (f,g) coefficients
  Memcheck:Cond
  src:keygen.c:4154 # fun:PQCLEAN_FALCON*_*_keygen
  fun:PQCLEAN_FALCON*_*_crypto_sign_keypair
}
```

MemSan

```
1  fun:PQCP_MLKEM_NATIVE_MLKEM*_C_poly_decompress_d*
2  fun:PQCP_MLKEM_NATIVE_MLKEM*_C_poly_reduce
3  fun:PQCP_MLKEM_NATIVE_MLKEM*_X86_64_poly_frommsg
4  fun:mlk_ct_memcmp
5  fun:mlk_ct_cmov_zero
6  fun:mlk_rej_uniform_scalar
7  fun:mlk_load32_littleendian
8  fun:PQCP_MLKEM_NATIVE_MLKEM*_C_poly_frommsg
9  fun:mlk_value_barrier_*
10 fun:PQCP_MLKEM_NATIVE_MLKEM*_X86_64_rej_uniform_avx2
11 fun:PQCP_MLKEM_NATIVE_MLKEM*_X86_64_enc_derand
12 fun:mlk_ct_cmask_neg_i16
13 fun:mlk_scalar_signed_to_unsigned_q
```

SNTRUP761 – Warning Analysis

- Recap of “Streamlined NTRU PRIME”: **SNTRUP761**
 - During decaps, Bob decrypts by multiplying sk with the ct received from Alice (two polynomials in a ring)
- **One rule:** The resulting polynomial from this decryption must be a 761-element that must have 286 coefficients set to 1 (weight = 286)
- This outcome is then passed on to a KDF to generate the shared secret

2.3.4 Streamlined NTRU Prime Core decryption

The following deterministic algorithm Decrypt maps Ciphertexts $\times$ SecretKeys to Inputs:

- Input $c \in \text{Ciphertexts}$ and $(f, v) \in \text{Short} \times \mathcal{R}/3$.
- Compute $3fc \in \mathcal{R}/q$. (One can multiply c by 3 and then by f , or multiply c by a precomputed $3f$, or multiply c by f and then by 3.)
- View each coefficient of $3fc$ in $\mathcal{R}/q$ as an integer between $-(q-1)/2$ and $(q-1)/2$, and then reduce modulo 3, obtaining a polynomial $e \in \mathcal{R}/3$.
- Multiply e by v in $\mathcal{R}/3$.
- Lift ev in $\mathcal{R}/3$ to a small polynomial $r' \in \mathcal{R}$.
- Output r' if r' has weight w . Otherwise output $(1, 1, \dots, 1, 0, 0, \dots, 0)$. (Implementors must be careful to avoid leaking secret information through side channels, and in particular must avoid implementing the weight test here as a branch.)

SNTRUP761 – Warning Analysis

liboqs/src/kem/ntruprime/pqclean_sntrup761_clean/crypto_core_wforcesntrup761.c

```
#include "crypto_core_wforcesntrup761.h"
```

```
#include "params.h"
```

```
#define int8 int8_t
```

```
#define int16 int16_t
```

```
#define uint16 uint16_t
```

```
#define uint32 uint32_t
```

```
typedef int8 small;
```

```
/* return -1 if x!=0; else return 0 */
```

```
static int int16_nonzero_mask(int16 x) {
```

```
    uint16 u = (uint16) x; /* 0, else 1...65535 */
```

```
    uint32 v = u; /* 0, else 1...65535 */
```

```
    v = ~v + 1; /* 0, else 2^32-65535...2^32-1 */
```

```
    v >>= 31; /* 0, else 1 */
```

```
    return -(int) v; /* 0, else -1 */
```

```
}
```

Uses bit-wise operations
to generate a mask

If input is 0 return 0,
else return -1

```
/* 0 if Weightw_is(r), else -1 */
```

```
static int Weightw_mask(const small *r) {
```

```
    int weight = 0;
```

```
    int i;
```

```
    for (i = 0; i < p; ++i) {
```

```
        weight += r[i] & 1;
```

```
    }
```

```
    return int16_nonzero_mask((int16) (weight - w));
```

```
}
```

```
/* out = in if bottom bits of in have weight w */
```

```
/* otherwise out = (1,1,...,1,0,0,...,0) */
```

```
int PQCLEAN_SNTRUP761_CLEAN_crypto_core_wforcesntrup761(unsigned char *outbytes, const unsigned char *inbytes) {
```

```
    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] = (small) (((in[i] ^ 1) & ~mask) ^ 1);
```

```
    }
```

```
    for (i = w; i < p; ++i) {
```

```
        out[i] = (small) (in[i] & ~mask);
```

```
    }
```

```
    return 0;
```

```
}
```

Calculates the weight by recursively
adding all coefficients

Subtracts result from 286

Returns 0 if result is 286 and -1 if
not

Checks whether the input's
weight = 286

If so, copies input into output,
else output = (1,1,...,1,0,0,...,0)

SNTRUP761 – Warning Analysis

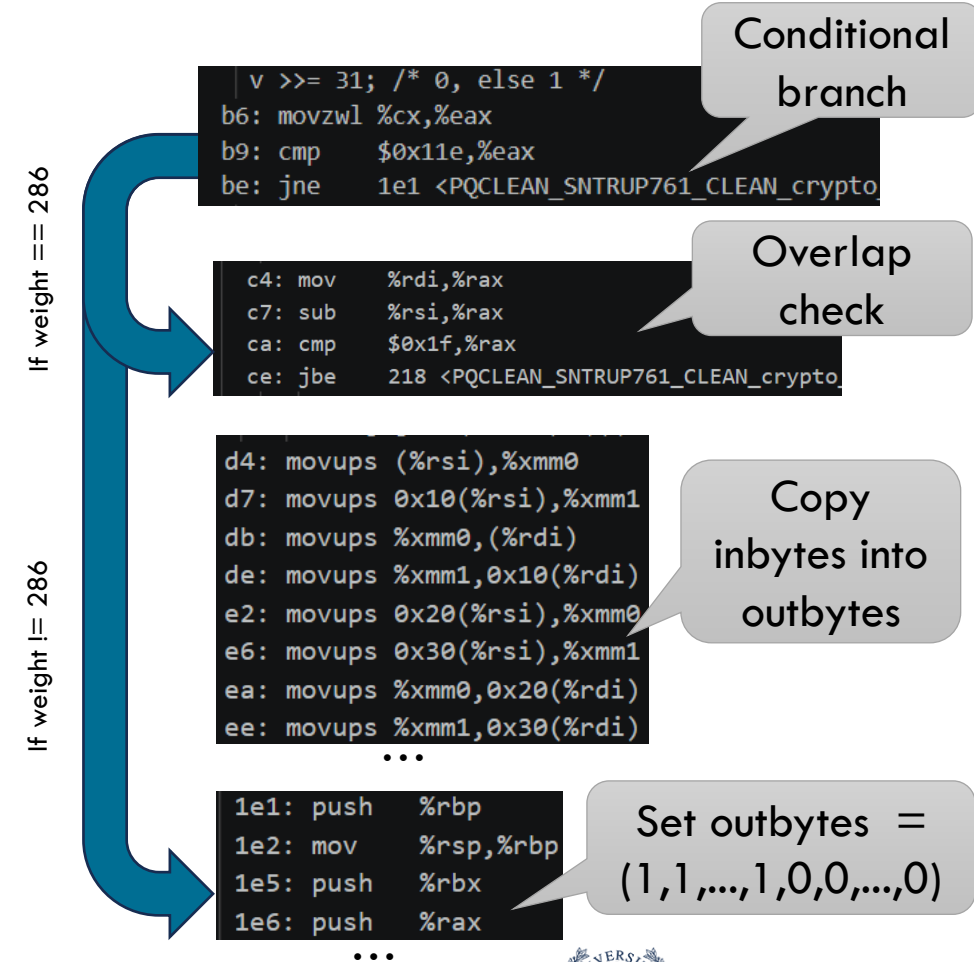
Source code (C)

```
/* return -1 if x!=0; else return 0 */
static int int16_nonzero_mask(int16 x) {
    uint16 u = (uint16) x; /* 0, else 1...65535 */
    uint32 v = u; /* 0, else 1...65535 */
    v = ~v + 1; /* 0, else 2^32-65535...2^32-1 */
    v >>= 31; /* 0, else 1 */
    return -(int) v; /* 0, else -1 */
}
```

Assembly (compiled with O0)

```
0000000000000120 <int16_nonzero_mask>:
static int int16_nonzero_mask(int16 x) {
120: push    %rbp
121: mov     %rsp,%rbp
124: mov     %di,%eax
127: mov     %ax,-0x2(%rbp)
    uint16 u = (uint16) x; /* 0, else 1...65535 */
12b: mov     -0x2(%rbp),%ax
12f: mov     %ax,-0x4(%rbp)
    uint32 v = u; /* 0, else 1...65535 */
133: movzwl  -0x4(%rbp),%eax
137: mov     %eax,-0x8(%rbp)
    v = ~v + 1; /* 0, else 2^32-65535...2^32-1 */
13a: mov     -0x8(%rbp),%eax
13d: xor     $0xffffffff,%eax
140: add     $0x1,%eax
143: mov     %eax,-0x8(%rbp)
    v >>= 31; /* 0, else 1 */
146: mov     -0x8(%rbp),%eax
149: shr     $0x1f,%eax
14c: mov     %eax,-0x8(%rbp)
    return -(int) v; /* 0, else -1 */
14f: xor     %eax,%eax
151: sub     -0x8(%rbp),%eax
154: pop     %rbp
155: ret
```

Assembly (compiled with O3)



SNTRUP761 – Warning Analysis

- The timing leak **does not provide information about sk directly**, but rather whether the **resultant weight after decaps is 286 or not**
- **Possible attack:** Crafting a ciphertext to learn sk based on the timing leak
 1. Design c' such that the weight of $sk \times c'$ equals 286 only if $g[0] = 1$
 2. Send c' and measure response time
 - Fast $\rightarrow$ weight was 286 $\rightarrow sk[0] = 1$
 - Slow $\rightarrow$ weight was not 286 $\rightarrow sk[0] = 0$
 3. Repeat crafting different c 's targeting the other positions within sk

SNTRUP761 – Warning Analysis

- Replaced PQClean code with public-domain OpenSSH implementation
 - Includes assembly instructions

```
crypto_int16 crypto_int16_nonzero_mask(crypto_int16 crypto_int16_x) {  
    #if defined(__GNUC__) && defined(__x86_64__)  
        crypto_int16 crypto_int16_q, crypto_int16_z;  
        __asm__ ("xorw %0,%0\n movw $-1,%1\n testw %2,%2\n cmovnew %1,%0" : "=r"(crypto_int16_z), "=r"(crypto_int16_q) : "r"(crypto_int16_x) : "cc");  
        return crypto_int16_z;  
    #elif defined(__GNUC__) && defined(__aarch64__)  
        crypto_int16 crypto_int16_z;  
        __asm__ ("tst %w1,65535\n csetm %w0,ne" : "=r"(crypto_int16_z) : "r"(crypto_int16_x) : "cc");  
        return crypto_int16_z;  
    #else  
        crypto_int16_x |= -crypto_int16_x;  
        return crypto_int16_negative_mask(crypto_int16_x);  
    #endif  
}
```

CI integration

- Workflow that executes tests on all compilation parameters described
 - Valgrind's Kyberslash patch is built on the CI container directly
- Enables user interaction for specific algorithm selection
 - Divide and conquer: More resources available
- Local execution is also available

The screenshot displays the 'ct-tooling' CI workflow interface. At the top, a header bar shows a green checkmark, the name 'ct-tooling', a subtitle 'ct-tooling #80: Manually run by pablo-gf', a 'ct-tooling' label, and a timestamp 'Jul 3, 12:52 PM GMT+2' with a duration of '30m 29s'. Below this, the interface is divided into three main sections. The left section, titled 'Algorithms to execute CT testing on (comma-separated) *', contains a text input field with the value 'ML-KEM-512,ML-KEM-768,ML-KEM-1024' and a green 'Run workflow' button. The middle section lists six job entries, each with a green checkmark and a description: 'valgrind-varlat (ML-KEM-512, gcc, generic, -O0)', 'valgrind-varlat (ML-KEM-512, gcc, generic, -O1)', 'valgrind-varlat (ML-KEM-512, gcc, generic, -O2)', 'valgrind-varlat (ML-KEM-512, gcc, generic, -O3)', 'valgrind-varlat (ML-KEM-512, gcc, generic, -Os)', and 'valgrind-varlat (ML-KEM-512, gcc, generic, -Ofast)'. The right section, titled 'ct-tooling.yml on: workflow_dispatch', shows a workflow diagram with a 'resolve-matrix' step (10s) leading to two matrix jobs: 'Matrix: memsan' (42 jobs completed) and 'Matrix: valgrind-varlat' (96 jobs completed).

Next steps

Triage algorithms based on warning counts



Analyze warnings and categorize false and true positives



Reach out to research groups and update implementations accordingly

A Constant-Time Analysis Framework for Post-Quantum Cryptography

Pablo Gutiérrez-Félix

Workshop on Secure Protocol Implementations in the Quantum Era

EUROS&P 2026, Lisbon, Portugal



pablogf@uma.es



linkedin.com/in/pablogutierrezfelix



github.com/pablo-gf