TRUST THE MATH, FEAR THE COMPILER:

How Optimisations undermine Cryptographic Software

René Meusel
FOSDEM 2026: /dev/random

ROHDE & SCHWARZ

Make ideas real

These slides may contain traces of assembly.

René Meusel

Rohde & Schwarz Cybersecurity

@reneme

Co-Maintaining the Botan* crypto library
Contributing to Open Source since 2011

* github.com/randombit/botan

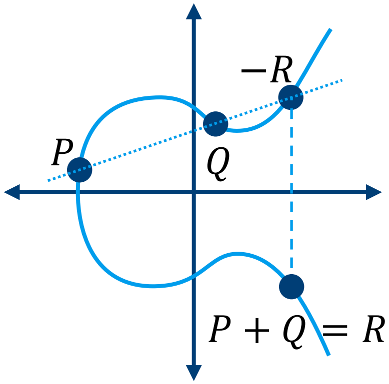
WHY DO WE TRUST THE MATH?

Prime Factorization Problem

$$15251262345256836781$$

$$= p1? \times p2?$$

(Elliptic Curve) Discrete Logarithm Problem



Learning with Errors Problem

$$b = A * s + e$$

All these problems are **believed** to be hard enough for cryptography.

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password

* * * * *

POST /auth

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

https://godbolt.org/z/1roz5sTjh

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password

* * * * *

POST /auth

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```


https://godbolt.org/z/1roz5sTjh

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password

* * * * *

POST /auth

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```

This is a Timing Side Channel!

Program behaviour depends on a secret

https://godbolt.org/z/1roz5sTjh

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password

* * * * *

POST /auth

fosdem26

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

"a" -> HTTP 401 after **1ms**

```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```


https://godbolt.org/z/1roz5sTjh

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password

* * * * *

POST /auth

fosdem26

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

```
"a"  -> HTTP 401 after 1ms
"b"  -> HTTP 401 after 1ms
"c"  -> HTTP 401 after 1ms
"d"  -> HTTP 401 after 1ms
"e"  -> HTTP 401 after 1ms
"f"  -> HTTP 401 after 2ms
```

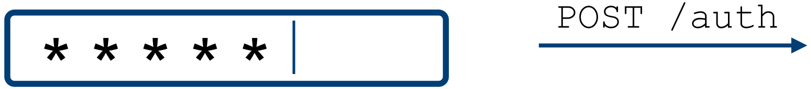
```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```


https://godbolt.org/z/1roz5sTjh

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password



fosdem26

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

```
"a"  -> HTTP 401 after 1ms
"b"  -> HTTP 401 after 1ms
"c"  -> HTTP 401 after 1ms
"d"  -> HTTP 401 after 1ms
"e"  -> HTTP 401 after 1ms
"f"  -> HTTP 401 after 2ms
"fa" -> HTTP 401 after 2ms
"fb" -> HTTP 401 after 2ms
"fc" -> HTTP 401 after 2ms
        ...
```

```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

Enter your password

* * * * *

POST /auth

```
"a"   -> HTTP 401 after 1ms
"b"   -> HTTP 401 after 1ms
"c"   -> HTTP 401 after 1ms
"d"   -> HTTP 401 after 1ms
"e"   -> HTTP 401 after 1ms
"f"   -> HTTP 401 after 2ms
"fa"  -> HTTP 401 after 2ms
"fb"  -> HTTP 401 after 2ms
"fc"  -> HTTP 401 after 2ms
              ...
"fosdem26" -> HTTP 200 after 8ms
```

fosdem26

```
if (equal(req.password, db.password))
    okay();
else
    unauthorized();
```

Let's ignore that we are saving plain passwords in our database

```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```


https://godbolt.org/z/1roz5sTjh

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

"Constant-time" implementation

```
bool equal(std::string pw1, std::string pw2)
{
    bool match = true;
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        // no side-channel: just keep comparing
        match = match && (a == b);
    }
    return match && pw1.size() == pw2.size();
}
```

Secret-dependent control flow

```
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```


https://godbolt.org/z/qqfvovnTx

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

"Constant-time" implementation

```cpp
bool equal(std::string pw1, std::string pw2)
{
    bool match = true;

    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        // no side-channel: just keep comparing
        match = match && (a == b);
    }


    return match && pw1.size() == pw2.size();
}
```

=> GCC 15.2, with -std=c++23 -O3

```asm
    [ loop setup ]
    mov     ecx, 1
.loop:
    test    cl, cl
    je      .way_out
    movzx   ecx, BYTE PTR [rdx]
    cmp     BYTE PTR [rax], cl
    sete    cl
    [ if needed, goto .loop ]
.way_out:
    cmp     r9, r8
    sete    al
    and     eax, ecx
    ret
```

The math is solid and trustworthy...

IT'S THE IMPLEMENTATIONS THAT FAIL!

"Constant-time" implementation

```
bool equal(std::string pw1, std::string pw2)
{
    bool match = true;

    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        // no side-channel: just keep comparing
        match = match && (a == b);
    }

    return match && pw1.size() == pw2.size();
}
```

=> GCC 15.2, with -std=c++23 -O3

```
    [ loop setup ]
    mov     ecx, 1
.loop:
    test    cl, cl
    je      .way_out
    movzx   ecx, BYTE PTR [rdx]
    cmp     BYTE PTR [rax], cl
    sete    cl
    [ if needed, goto .loop ]
.way_out:
    cmp     r9, r8
    sete    al
    and     eax, ecx
    ret
```

```
match = (a == b);
```


https://godbolt.org/z/qqfvovnTx

IT'S THE IMPLEMENTATIONS THAT FAIL!

"Constant-time" implementation

```
bool equal(std::string pw1, std::string pw2)
{
    bool match = true;

    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        // no side-channel: just keep comparing
        match = match && (a == b);
    }

    return match && pw1.size() == pw2.size();
}
```

=> GCC 15.2, with -std=c++23 -O3

```
    [ loop setup ]
    mov     ecx, 1
.loop:
    test    cl, cl                      if (!match)
    je      .way_out                        break;
    movzx   ecx, BYTE PTR [rdx]
    cmp     BYTE PTR [rax], cl          match = (a == b);
    sete    cl
    [ if needed, goto .loop ]
.way_out:
    cmp     r9, r8
    sete    al
    and     eax, ecx
    ret
```

GCC "optimized" the side-channel back into our safe implementation!

Use `uint32_t` instead of `bool`

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
bool equal(std::string pw1, std::string pw2)
{
    bool match = true;
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        match = match && (a == b);
    }
    return match && pw1.size() == pw2.size();
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    return ((~v & (v - 1)) >> 31) - 1;
}
```

Optimizing boolean logic is easy for the compiler.

... let’s try to hide booleans behind ordinary integers.

https://godbolt.org/z/cKGxP6T75

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        match = match && (a == b);
    }
    return match && pw1.size() == pw2.size();
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    return ((~v & (v - 1)) >> 31) - 1;
}
```

Optimizing boolean logic is easy for the compiler.

... let’s try to hide booleans behind ordinary integers.

https://godbolt.org/z/cKGxP6T75

Use uint32_t instead of bool

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return mask & expand(pw1.size() == pw2.size());
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    return ((~v & (v - 1)) >> 31) - 1;
}
```

Optimizing boolean logic is easy for the compiler.

... let’s try to hide booleans behind ordinary integers.

https://godbolt.org/z/cKGxP6T75

Use `uint32_t` instead of `bool`

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {


        mask = mask & expand(a == b);


    }

    return match & expand(pw1.size() == pw2.size());
}
```

```
    [ loop setup ]
    mov     ecx, -1
.loop:
    movzx   esi, BYTE PTR [rdx]
    cmp     BYTE PTR [rax], sil
    setne   sil
    movzx   esi, sil
    sub     esi, 1
    and     ecx, esi
    [ if needed, goto .loop ]
.after_loop:
    xor     eax, eax
    cmp     r9, r10
    setne   al
    sub     eax, 1
    test    eax, ecx
    setne   al
    ret
```


https://godbolt.org/z/cKGxP6T75

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return match & expand(pw1.size() == pw2.size());
}
```

```
    [ loop setup ]
    mov     ecx, -1
.loop:
    movzx   esi, BYTE PTR [rdx]
    cmp     BYTE PTR [rax], sil
    setne   sil
    movzx   esi, sil
    sub     esi, 1
    and     ecx, esi
    [ if needed, goto .loop ]
.after_loop:
    xor     eax, eax
    cmp     r9, r10
    setne   al
    sub     eax, 1
    test    eax, ecx
    setne   al
    ret
```

esi = (a != b)

mask &= (esi−1)

((~v&(v-1))>>31)-1

Technically, okay.
But GCC continues to reason about the boolean!

https://godbolt.org/z/cKGxP6T75

Use advanced tricks to obfuscate integer value ranges

HIDING VALUES AND SEMANTICS FROM THE COMPILER

Due to inlining, the compiler might know that the parameter is actually a boolean!

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return mask & expand(pw1.size() == pw2.size());
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    return ((~v & (v - 1)) >> 31) - 1;
}
```


https://godbolt.org/z/63ocb5frv

HIDING VALUES AND SEMANTICS FROM THE COMPILER

Due to inlining, the compiler might know that the parameter is actually a boolean!

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return mask & expand(pw1.size() == pw2.size());
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    const uint32_t v2 = obfuscate(v);
    return ((~v2 & (v2 - 1)) >> 31) - 1;
}
```


https://godbolt.org/z/63ocb5frv

Use advanced tricks to obfuscate integer value ranges

HIDING VALUES AND SEMANTICS FROM THE COMPILER

Due to inlining, the compiler might know that the parameter is actually a boolean!

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return mask & expand(pw1.size() == pw2.size());
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    const uint32_t v2 = obfuscate(v);
    return ((~v2 & (v2 - 1)) >> 31) - 1;
}
```

*Extracts the most-significant **bit**. Compilers might recognize that as boolean!*

Compilers infer plausible integer ranges and use that for optimization.

Use advanced tricks to obfuscate integer value ranges

HIDING VALUES AND SEMANTICS FROM THE COMPILER

Due to inlining, the compiler might know that the parameter is actually a boolean!

```
bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return mask & expand(pw1.size() == pw2.size());
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    const uint32_t v2 = obfuscate(v);
    return obfuscate((~v2 & (v2 - 1)) >> 31) - 1;
}
```

*Extracts the most-significant **bit**.*
Compilers might recognize that as boolean!

Compilers infer plausible integer ranges and use that for optimization.

https://godbolt.org/z/63ocb5frv

Use advanced tricks to obfuscate integer value ranges

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
a == b

expand(uint32_t v)
{
  ((~v&(v-1))>>31)-1
}

mask &= ...
```

```
    [ preamble, loop setup ]
.loop:
    movzx    eax, BYTE PTR [r8]
    xor      edx, edx
    cmp      BYTE PTR [rcx], al
    sete     dl
    mov      eax, edx
    sub      edx, 1
    not      eax
    and      eax, edx
    shr      eax, 31
    sub      eax, 1
    and      r9d, eax
    [ if needed, goto .loop ]
.way_out:
    [ epilogue ]
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    const uint32_t v2 = obfuscate(v);
    return obfuscate((~v2 & (v2 - 1)) >> 31) - 1;
}
```

Yay! No more optimization!

https://godbolt.org/z/63ocb5frv

Use advanced tricks to obfuscate integer value ranges

HIDING VALUES AND SEMANTICS FROM THE COMPILER

```
/// Hide the value of @p v from the compiler
/// and return it unchanged.
///
inline uint32_t obfuscate(uint32_t v)
{
    asm("" : "+r"(v) :);
    return v;
}
```

```
/// @returns 0x00000000 if v is zero,
///          0xFFFFFFFF otherwise
///
/// Uses only bitwise logic, no branches.
///
inline uint32_t expand(uint32_t v)
{
    const uint32_t v2 = obfuscate(v);
    return obfuscate((~v2 & (v2 - 1)) >> 31) - 1;
}
```

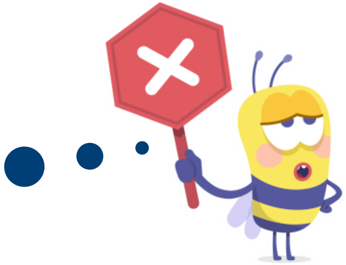
Compilers cannot reason about values produced by inline assembly.

https://godbolt.org/z/63ocb5frv

```cpp
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```

```cpp
bool equal(std::string pw1, std::string pw2)
{
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        if (a != b)
        {
            // early return for efficiency!
            return false;
        }
    }
    return pw1.size() == pw2.size();
}
```

This is a mess! *How to maintain this?*



```cpp
inline uint32_t obfuscate(uint32_t v)
{
    asm("" : "+r"(v) :);
    return v;
}


inline uint32_t expand(uint32_t v)
{
    const uint32_t v2 = obfuscate(v);
    return obfuscate((~v2 & (v2 - 1)) >> 31) - 1;
}


bool equal(std::string pw1, std::string pw2)
{
    uint32_t mask = expand(true);
    for (auto [a, b] : std::views::zip(pw1, pw2))
    {
        mask = mask & expand(a == b);
    }
    return mask & expand(pw1.size() == pw2.size());
}
```

SURPRISINGLY: VALGRIND CAN HELP

Valgrind warns if program behaviour depends on “uninitialized memory”.

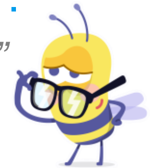
```
const std::string pw = get_password_from_user();

VALGRIND_MAKE_MEM_UNDEFINED(pw.data(), pw.size());
const bool ok = bad_equal(pw, the_password);
VALGRIND_MAKE_MEM_DEFINED(&ok, sizeof(ok));

if(ok)  // this branch is fine!
   okay();
else
   unauthorized();
```


github.com/agl/ctgrind

SURPRISINGLY: VALGRIND CAN HELP

Valgrind warns if program behaviour depends on “uninitialized memory”.

... let’s just claim that our “secret memory” is “uninitialized”.



```
const std::string pw = get_password_from_user();

VALGRIND_MAKE_MEM_UNDEFINED(pw.data(), pw.size());
const bool ok = bad_equal(pw, the_password);
VALGRIND_MAKE_MEM_DEFINED(&ok, sizeof(ok));

if(ok)  // this branch is fine!
   okay();
else
   unauthorized();
```

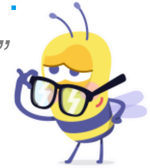
```
#> valgrind ./check_password fosdem2026
[...]
Conditional jump or move depends on uninitialised value(s)
   at: bad_equal(...) (equal.cpp:41)
   by: main (equal.cpp:68)
```


github.com/agl/ctgrind

SURPRISINGLY: VALGRIND CAN HELP

Valgrind warns if program behaviour depends on “uninitialized memory”.

... let’s just claim that our “secret memory” is “uninitialized”.



```
const std::string pw = get_password_from_user();

VALGRIND_MAKE_MEM_UNDEFINED(pw.data(), pw.size());
const bool ok = bad_equal(pw, the_password);
VALGRIND_MAKE_MEM_DEFINED(&ok, sizeof(ok));

if(ok)  // this branch is fine!
   okay();
else
   unauthorized();
```

Valgrind even finds that results were calculated from “uninitialized” / “secret” data.

... we have to “declassify” inferred values before using them to determine program flow.

```
#> valgrind ./check_password fosdem2026
[...]
Conditional jump or move depends on uninitialised value(s)
   at: bad_equal(...) (equal.cpp:41)
   by: main (equal.cpp:68)
```

SURPRISINGLY: VALGRIND CAN HELP

Valgrind warns if program behaviour
depen

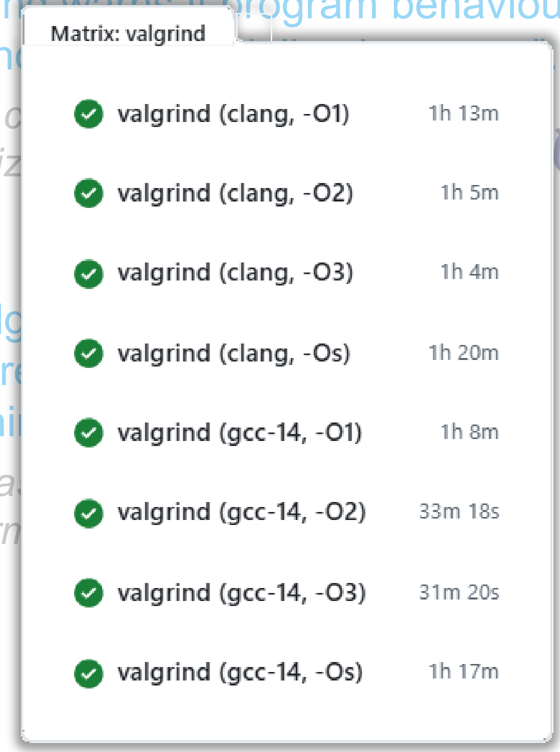
... let’s just c
is “uninitializ

```
const std::string pw = get_password_from_user();

VALGRIND_MAKE_MEM_UNDEFINED(pw.data(), pw.size());
const bool ok = bad_equal(pw, the_password);
VALGRIND_MAKE_MEM_DEFINED(&ok, sizeof(ok));

if(ok)  // this branch is fine!
   okay();
else
   unauthorized();
```

Valg
were
“uni

... we have to “decla
using them to determ

```
#> valgrind ./check_password fosdem2026
[...]
Conditional jump or move depends on uninitialised value(s)
   at: bad_equal(...) (equal.cpp:41)
   by: main (equal.cpp:68)
```



Botan’s nightly tests

github.com/agl/ctgrind

SUMMARY

- **Compilers make code *efficient*.**
  But they don't take other qualitative requirements into account.

- **Security products must always keep the entire system in mind.**
  Depending on the hardware, even individual instructions might have side-channels.[1]

- **Implementing cryptography is tough, not just because of the math.**
  Don't do it on your own, contribute to existing projects if you want to learn.

=> GCC 15.2, with -std=c++23 -O3

```
  [ loop setup ]
  mov     ecx, 1
.loop:
  test    cl, cl                    if (!match)
  je      .way_out                      break;
  movzx   ecx, BYTE PTR [rdx]
  cmp     BYTE PTR [rax], cl
  sete    cl
  [ if needed, goto .loop ]
.way_out:
  cmp     r9, r8
  sete    al
  and     eax, ecx
  ret
```

[1] https://kyberslash.cr.yp.to/papers.html