Cl[ia]ngWrap: Wrapping libraries with libclang

Kevin Huck, University of Oregon

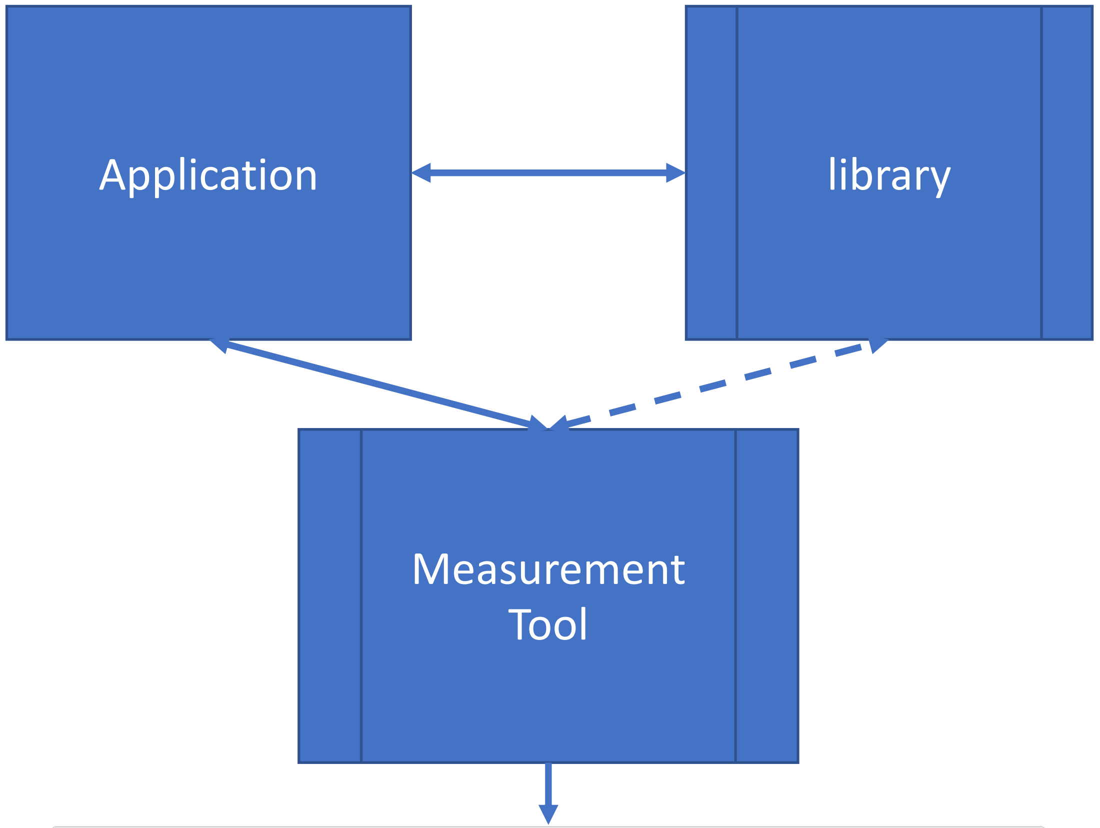
h/t Alister Johnson, University of Oregon

Quick poll

- C++ familiarity
  - It’s roots as “C with classes”
  - First C++ compilers just did C++ → C conversion, then compilation of the C
- Template comfort level
  - Super powerful, yet super confusing
  - Enables generic “container” classes…among other things
- Systems programming in general
  - How *nix based systems are built
  - …with C, by and large. It’s the lowest common denominator

Motivation

- Want to measure application
- Application uses a library that is not instrumented
- Sampling is an option (not discussed here)
  - May not have debug symbols
- …What about measuring every call into that external library?
- …Without modifying the application *or* the library?



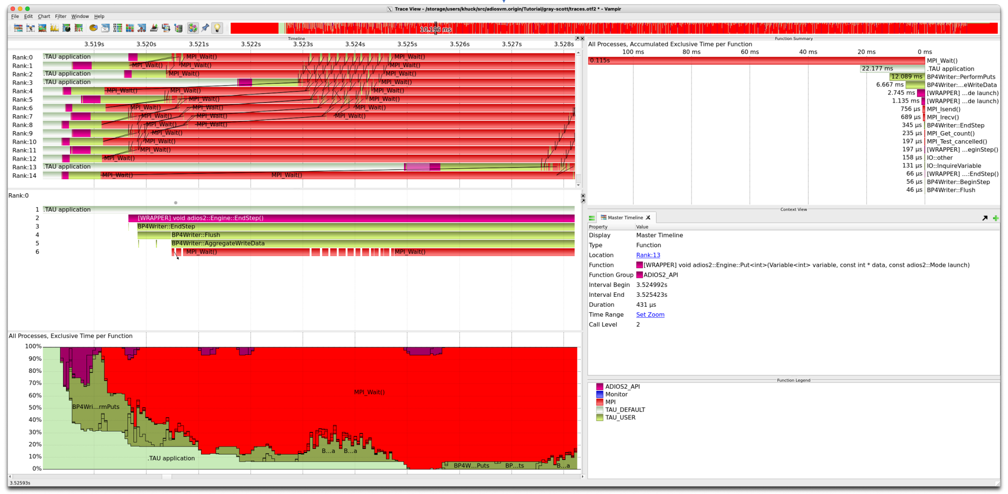


Figure: ADIOS2 tutorial example, measured by TAU, visualized with Vampir Trace Viewer

System tricks to wrap libraries

- A symbol can only be defined once in an executable/application
- Weak symbols
  - “Default” implementation of a function, can be overridden at link/runtime
  - Requires cooperation of the library developer (*very* rare) – MPI is an example
- Static linking
  - Linker option: `--wrap symbol`
  - Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to "__wrap_symbol". Any undefined reference to "__real_symbol" will be resolved to symbol. (source: man page for `ld`)
  - Have to specify for every…single…function…in…the…API...
- Dynamic linking tricks (linker order, LD_PRELOAD)

Existing (automated) solutions (examples)

- TAU wrapper
  - Uses PDT (Program Database Toolkit), based on EDG 4.0 parser
  - Lindlan, K. A., Cuny, J., Malony, A. D., Shende, S., Mohr, B., Rivenburgh, R., & Rasmussen, C. (2000, November). A tool framework for static and dynamic analysis of object-oriented software with templates. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing* (pp. 49-49). IEEE.
  - https://www.cs.uoregon.edu/research/tau/downloads.php
  - Depends on ability to parse the API header
- Gotcha
  - Poliakoff D., LeGendre M. (2019) Gotcha: An Function-Wrapping Interface for HPC Tools. In: *Bhatele A., Boehme D., Levine J., Malony A., Schulz M. (eds) Programming and Performance Visualization Tools*. ESPT 2017, ESPT 2018, VPA 2017, VPA 2018. Lecture Notes in Computer Science, vol 11027. Springer, Cham. https://doi.org/10.1007/978-3-030-17872-7_11
  - https://github.com/LLNL/GOTCHA
  - Depends on the ability to extract symbols from library
  - User has to write…all the code.

API Compatibility vs. ABI Compatibility

Sidebar…

- API – Application Programming Interface
  - What the human programs against
  - Header file, documentation
- ABI – Application Binary Interface
  - What the compiler generates
  - Actual symbols and function types in the library – the “*mangled*” name
- ABI != API
- For (lots) more, see https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Dialect-Options.html and https://gcc.gnu.org/onlinedocs/libstdc++/manual/abi.html (and that’s just GCC’s documentation…but most compilers depend on the system installed libc, libstdc, libc++ libstdc++…which for Linux is GCC.) Distro vendors also adress this problem, see https://developers.redhat.com/blog/2015/02/05/gcc5-and-the-c11-abi/)

Problem with existing solutions

- Assumption: C binding rules
  - Symbol name == function name …no? (see previous slide)
- Doesn’t take into consideration C++ name mangling
  - C++ requires name “mangling” to allow for function overloading

  ```
  template <> std::string Attribute<char>::Name() const;
  _ZNK6adios29AttributeIcE4NameB5cxx11Ev
  template <> std::string Attribute<signed char>::Name() const;
  _ZNK6adios29AttributeIaE4NameB5cxx11Ev
  ```

  - Hidden/implied “this” pointer for member functions (conceptually like “self” in Python)
- Fortran is another issue, but an easier case to manage
  - Symbols are case-insensitive in the language, but symbols are not!
  - May require include 0, 1, 2 underscores at the beginning of the symbol
- Decent overview of mangling:
  - https://en.wikipedia.org/wiki/Name_mangling
- ABI has to match perfectly (-ish?), or function wrapper won’t get called!
- Mangled name has to match perfectly, or wrapper will call the wrong library function!

  (hint: bad things happen)

Proposed solution: libclang based approach

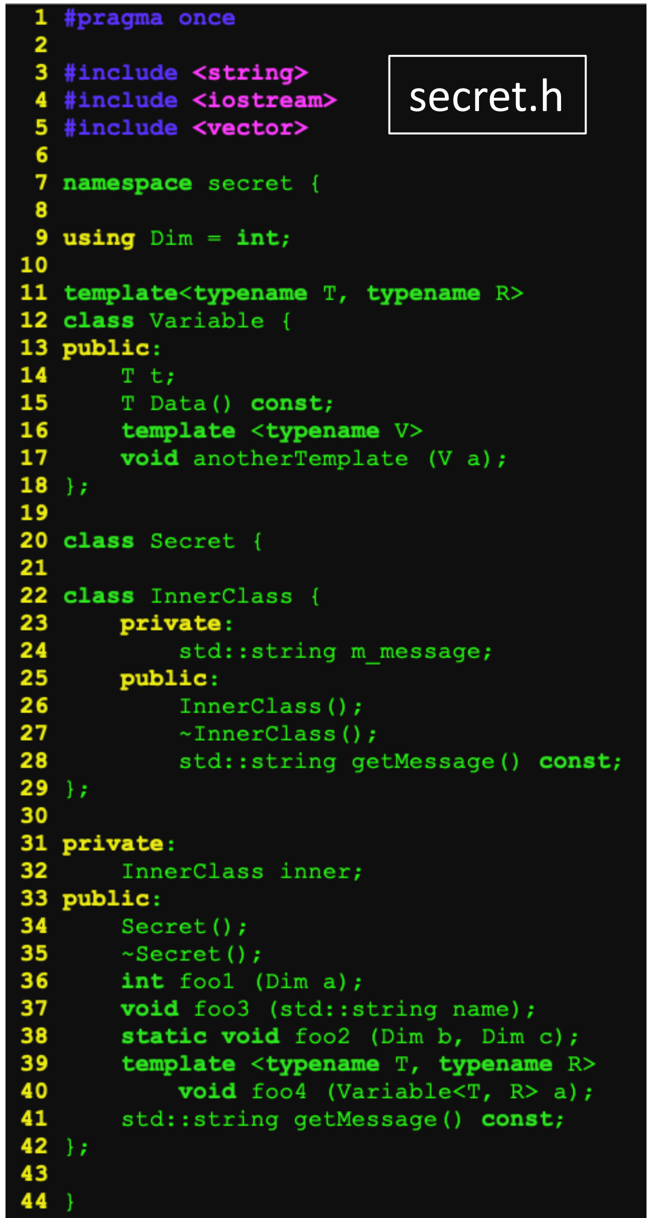
- Extract symbols from library using `nm` utility (limited to functions in a user-specified C++ namespace)
- Demangle symbols in library, find function name and number of arguments, put them in a map
- Parse API header AST using libclang (parser for clang++)
- *Could* mangle AST names…but "that way be dragons" (no "standard" algorithm for mangling)
- Match header declarations with demangled symbols (harder than it sounds, unfortunately)
  - Make const style consistent (`const Foo &` or `Foo const &`) …compiler's choice, sadly
  - Expand aliases (`using Foo = std::vector<std::string>;`)
  - Standardize `std::string` usage (it gets expanded in the symbol name – see box below)
  - Allow the user to specify any other typename mappings as necessary (`MPI_Comm` -> `ompi_communicator_t*`)
  - Match function name and number of arguments…. (improves chances of "perfect" match)
  - …*then* do a string alignment search to find the "best" match (with heuristic based scores)
- Generate library to wrap API
  - Relies on `dlopen()`,`dlsym()` to call actual functions
- LD_PRELOAD or link with wrapper library (instead of / before actual library) – remember, symbols can only be defined *once*

`std::string` is:

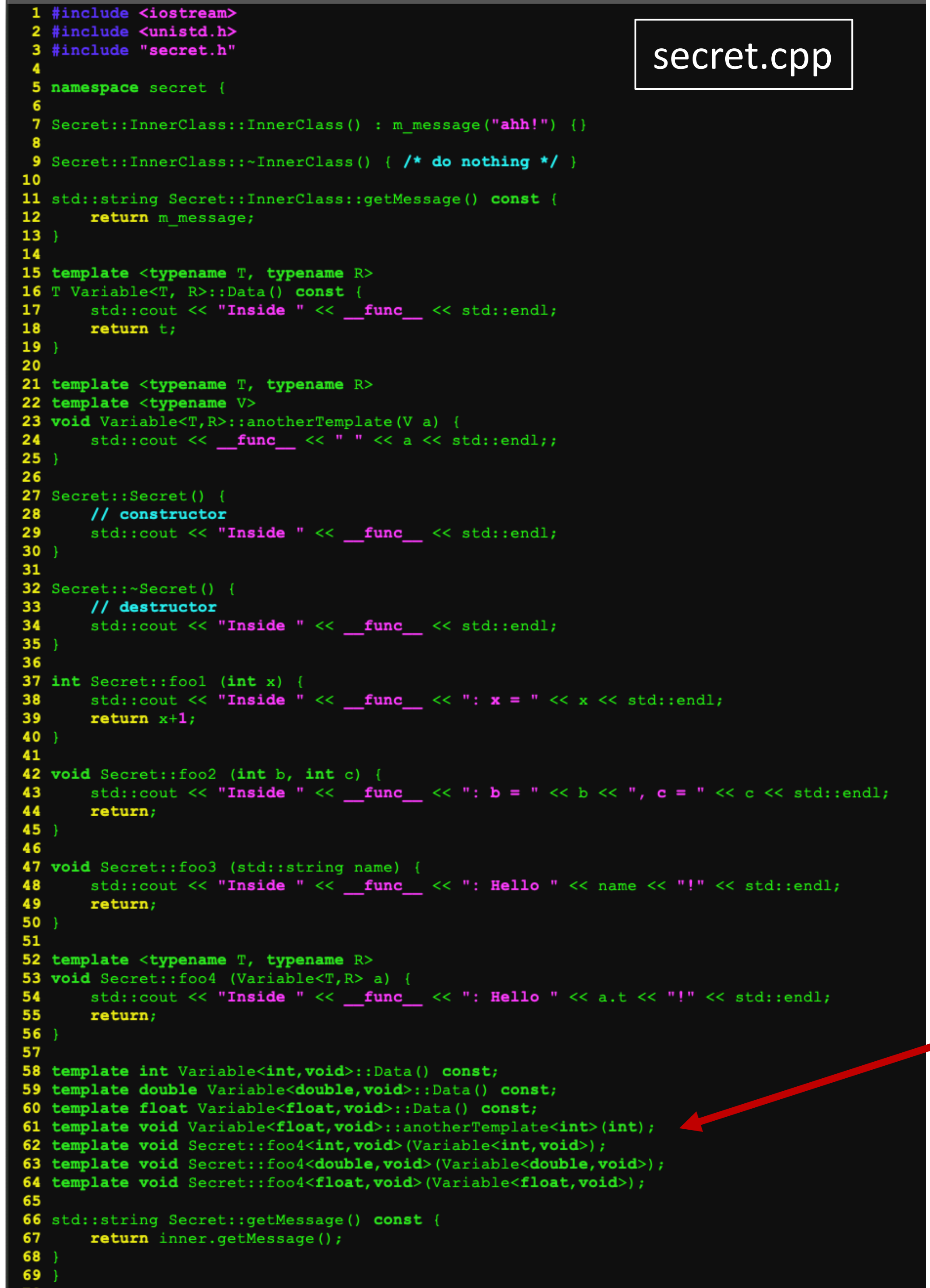
`std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >` …or…

`std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::allocator<char> >`

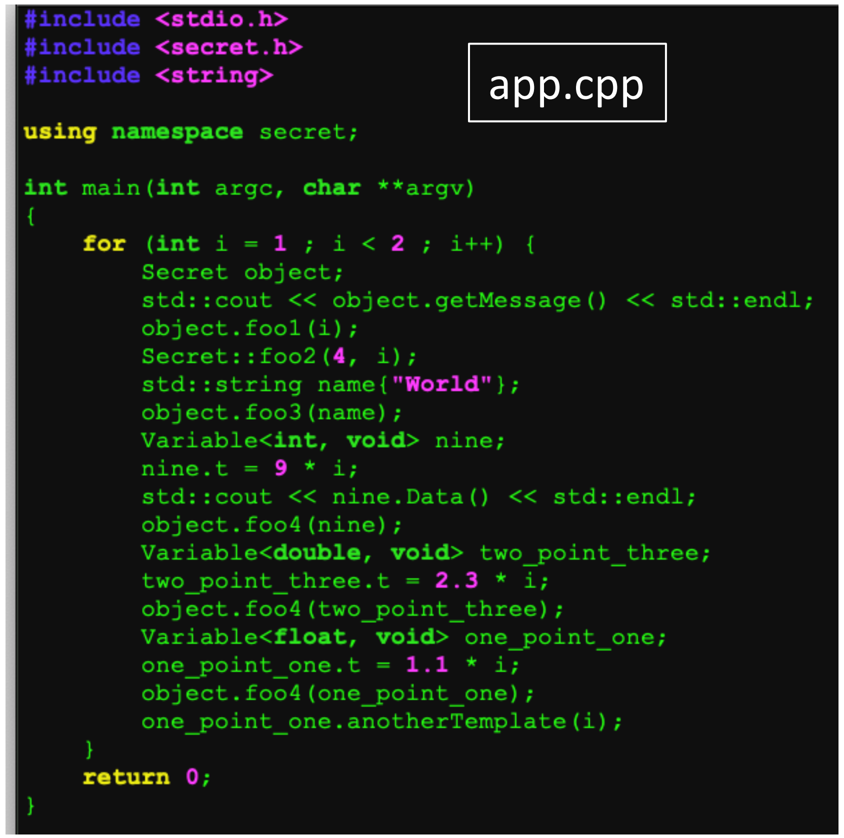
Example


secret.h

```cpp
#pragma once

#include <string>
#include <iostream>
#include <vector>

namespace secret {

using Dim = int;

template<typename T, typename R>
class Variable {
public:
    T t;
    T Data() const;
    template <typename V>
    void anotherTemplate (V a);
};

class Secret {

class InnerClass {
    private:
        std::string m_message;
    public:
        InnerClass();
        ~InnerClass();
        std::string getMessage() const;
};

private:
    InnerClass inner;
public:
    Secret();
    ~Secret();
    int foo1 (Dim a);
    void foo3 (std::string name);
    static void foo2 (Dim b, Dim c);
    template <typename T, typename R>
        void foo4 (Variable<T, R> a);
    std::string getMessage() const;
};

}
```


secret.cpp

```cpp
#include <iostream>
#include <unistd.h>
#include "secret.h"

namespace secret {

Secret::InnerClass::InnerClass() : m_message("ahh!") {}

Secret::InnerClass::~InnerClass() { /* do nothing */ }

std::string Secret::InnerClass::getMessage() const {
    return m_message;
}

template <typename T, typename R>
T Variable<T, R>::Data() const {
    std::cout << "Inside " << __func__ << std::endl;
    return t;
}

template <typename T, typename R>
template <typename V>
void Variable<T,R>::anotherTemplate(V a) {
    std::cout << __func__ << " " << a << std::endl;;
}

Secret::Secret() {
    // constructor
    std::cout << "Inside " << __func__ << std::endl;
}

Secret::~Secret() {
    // destructor
    std::cout << "Inside " << __func__ << std::endl;
}

int Secret::foo1 (int x) {
    std::cout << "Inside " << __func__ << ": x = " << x << std::endl;
    return x+1;
}

void Secret::foo2 (int b, int c) {
    std::cout << "Inside " << __func__ << ": b = " << b << ", c = " << c << std::endl;
    return;
}

void Secret::foo3 (std::string name) {
    std::cout << "Inside " << __func__ << ": Hello " << name << "!" << std::endl;
    return;
}

template <typename T, typename R>
void Secret::foo4 (Variable<T,R> a) {
    std::cout << "Inside " << __func__ << ": Hello " << a.t << "!" << std::endl;
    return;
}

template int Variable<int,void>::Data() const;
template double Variable<double,void>::Data() const;
template float Variable<float,void>::Data() const;
template void Variable<float,void>::anotherTemplate<int>(int);
template void Secret::foo4<int,void>(Variable<int,void>);
template void Secret::foo4<double,void>(Variable<double,void>);
template void Secret::foo4<float,void>(Variable<float,void>);

std::string Secret::getMessage() const {
    return inner.getMessage();
}
}
```

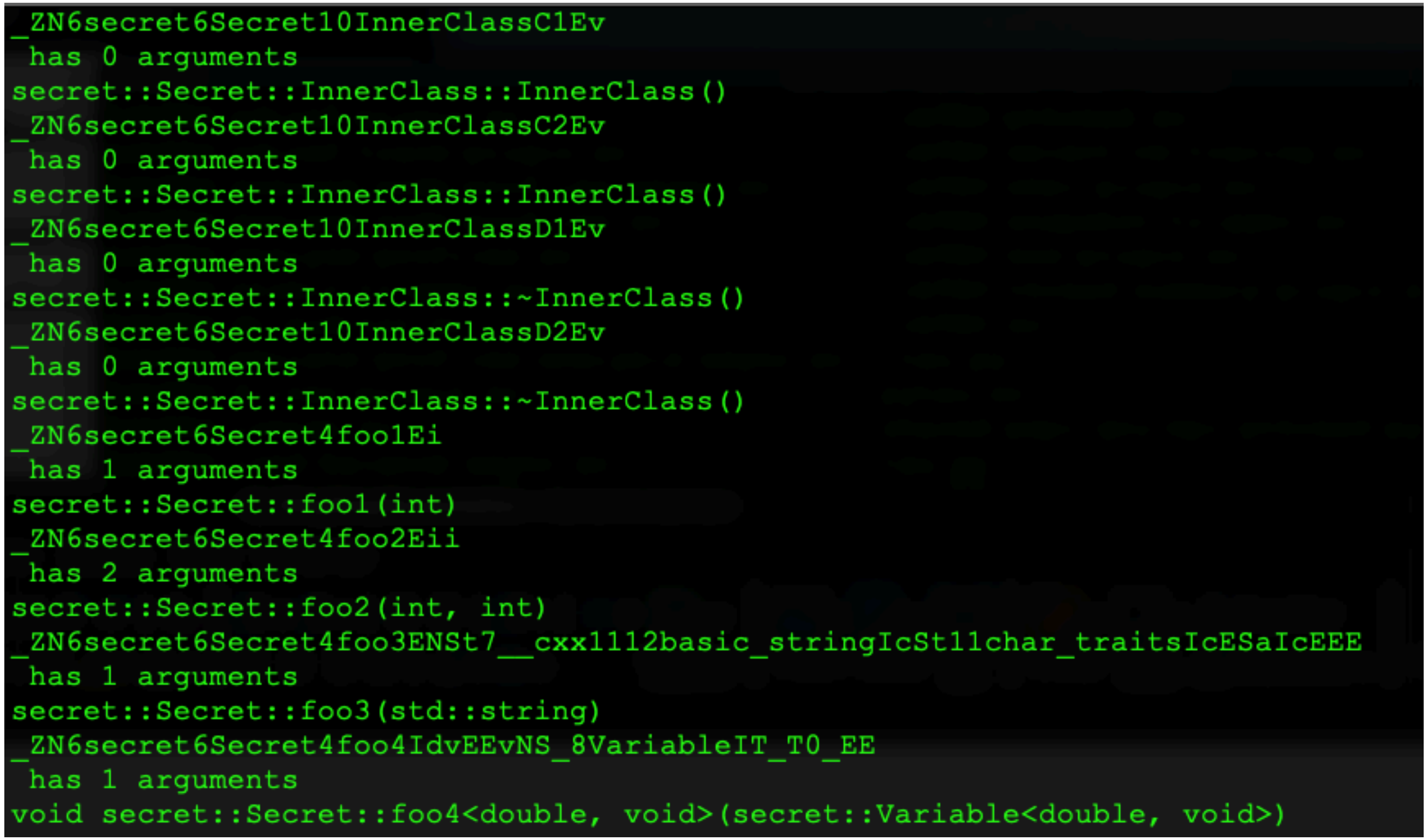

app.cpp

```cpp
#include <stdio.h>
#include <secret.h>
#include <string>

using namespace secret;

int main(int argc, char **argv)
{
    for (int i = 1 ; i < 2 ; i++) {
        Secret object;
        std::cout << object.getMessage() << std::endl;
        object.foo1(i);
        Secret::foo2(4, i);
        std::string name{"World"};
        object.foo3(name);
        Variable<int, void> nine;
        nine.t = 9 * i;
        std::cout << nine.Data() << std::endl;
        object.foo4(nine);
        Variable<double, void> two_point_three;
        two_point_three.t = 2.3 * i;
        object.foo4(two_point_three);
        Variable<float, void> one_point_one;
        one_point_one.t = 1.1 * i;
        object.foo4(one_point_one);
        one_point_one.anotherTemplate(i);
    }
    return 0;
}
```

Template instantiations/specializations

Example: Symbols extracted from library

```
nm libsecret.so | grep " [TW] " | grep "_Z"
```

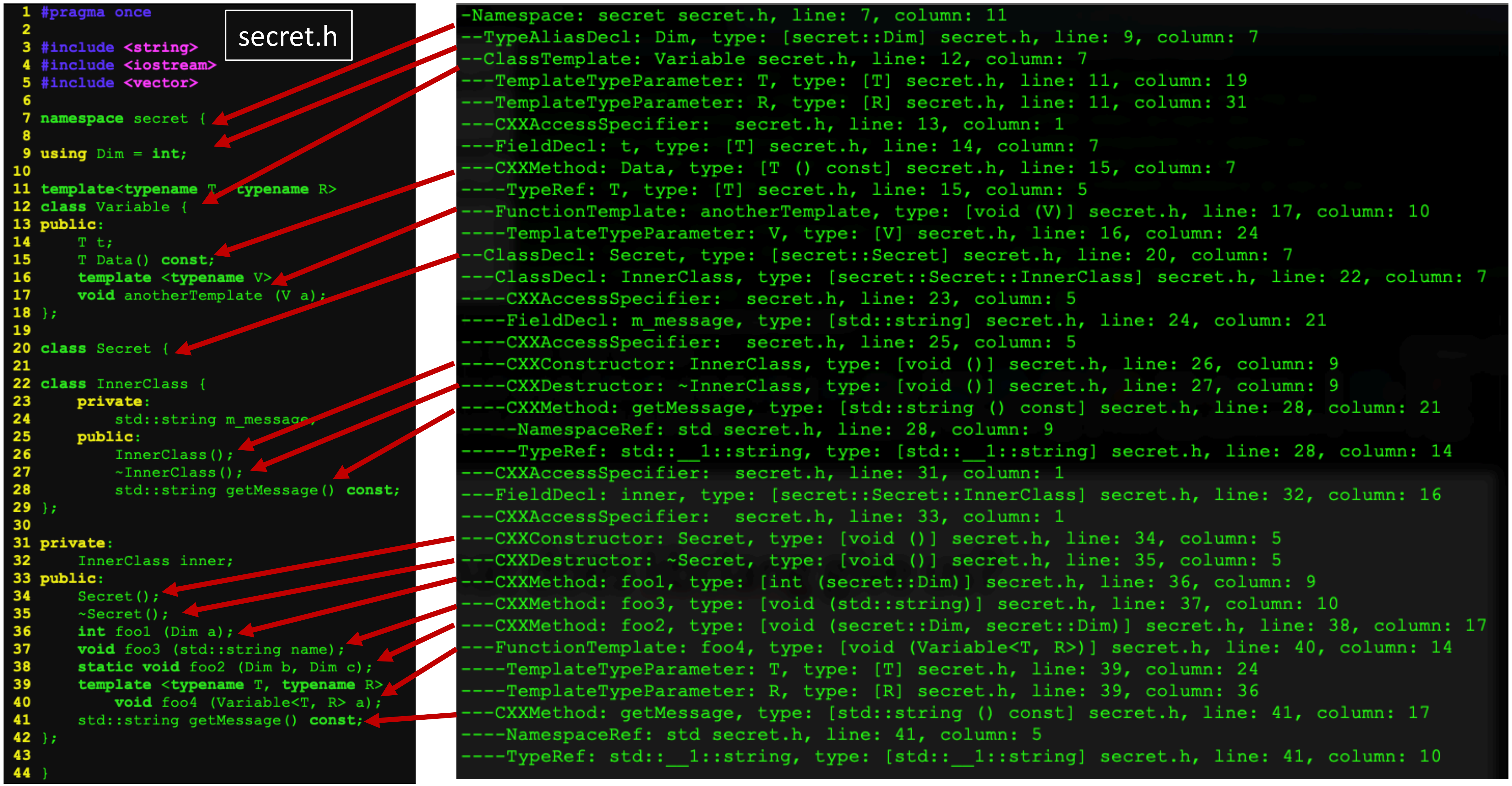


```
_ZN6secret6Secret10InnerClassC1Ev
 has 0 arguments
secret::Secret::InnerClass::InnerClass()
_ZN6secret6Secret10InnerClassC2Ev
 has 0 arguments
secret::Secret::InnerClass::InnerClass()
_ZN6secret6Secret10InnerClassD1Ev
 has 0 arguments
secret::Secret::InnerClass::~InnerClass()
_ZN6secret6Secret10InnerClassD2Ev
 has 0 arguments
secret::Secret::InnerClass::~InnerClass()
_ZN6secret6Secret4foo1Ei
 has 1 arguments
secret::Secret::foo1(int)
_ZN6secret6Secret4foo2Eii
 has 2 arguments
secret::Secret::foo2(int, int)
_ZN6secret6Secret4foo3ENSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEE
 has 1 arguments
secret::Secret::foo3(std::string)
_ZN6secret6Secret4foo4IdvEEvNS_8VariableIT_T0_EE
 has 1 arguments
void secret::Secret::foo4<double, void>(secret::Variable<double, void>)
```

(and so on...)

What does libclang give us?

(and much more…)



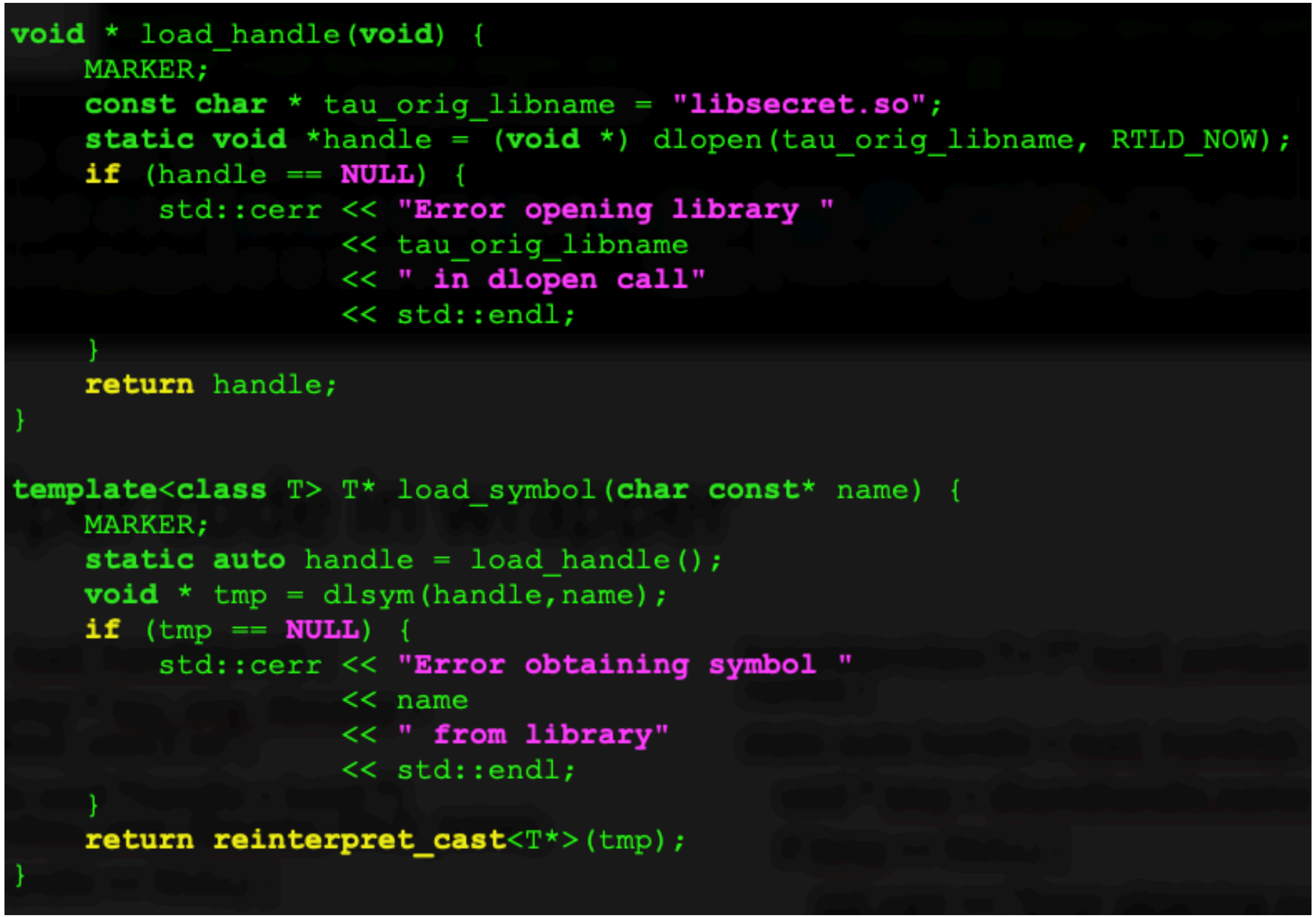
secret.h

```
 1 #pragma once
 2
 3 #include <string>
 4 #include <iostream>
 5 #include <vector>
 6
 7 namespace secret {
 8
 9 using Dim = int;
10
11 template<typename T, typename R>
12 class Variable {
13 public:
14     T t;
15     T Data() const;
16     template <typename V>
17     void anotherTemplate (V a);
18 };
19
20 class Secret {
21
22 class InnerClass {
23     private:
24         std::string m_message;
25     public:
26         InnerClass();
27         ~InnerClass();
28         std::string getMessage() const;
29 };
30
31 private:
32     InnerClass inner;
33 public:
34     Secret();
35     ~Secret();
36     int foo1 (Dim a);
37     void foo3 (std::string name);
38     static void foo2 (Dim b, Dim c);
39     template <typename T, typename R>
40         void foo4 (Variable<T, R> a);
41     std::string getMessage() const;
42 };
43
44 }
```

```
-Namespace: secret secret.h, line: 7, column: 11
--TypeAliasDecl: Dim, type: [secret::Dim] secret.h, line: 9, column: 7
--ClassTemplate: Variable secret.h, line: 12, column: 7
---TemplateTypeParameter: T, type: [T] secret.h, line: 11, column: 19
---TemplateTypeParameter: R, type: [R] secret.h, line: 11, column: 31
---CXXAccessSpecifier:  secret.h, line: 13, column: 1
---FieldDecl: t, type: [T] secret.h, line: 14, column: 7
---CXXMethod: Data, type: [T () const] secret.h, line: 15, column: 7
----TypeRef: T, type: [T] secret.h, line: 15, column: 5
---FunctionTemplate: anotherTemplate, type: [void (V)] secret.h, line: 17, column: 10
----TemplateTypeParameter: V, type: [V] secret.h, line: 16, column: 24
--ClassDecl: Secret, type: [secret::Secret] secret.h, line: 20, column: 7
---ClassDecl: InnerClass, type: [secret::Secret::InnerClass] secret.h, line: 22, column: 7
----CXXAccessSpecifier:  secret.h, line: 23, column: 5
----FieldDecl: m_message, type: [std::string] secret.h, line: 24, column: 21
----CXXAccessSpecifier:  secret.h, line: 25, column: 5
----CXXConstructor: InnerClass, type: [void ()] secret.h, line: 26, column: 9
----CXXDestructor: ~InnerClass, type: [void ()] secret.h, line: 27, column: 9
----CXXMethod: getMessage, type: [std::string () const] secret.h, line: 28, column: 21
-----NamespaceRef: std secret.h, line: 28, column: 9
-----TypeRef: std::__1::string, type: [std::__1::string] secret.h, line: 28, column: 14
---CXXAccessSpecifier:  secret.h, line: 31, column: 1
---FieldDecl: inner, type: [secret::Secret::InnerClass] secret.h, line: 32, column: 16
---CXXAccessSpecifier:  secret.h, line: 33, column: 1
---CXXConstructor: Secret, type: [void ()] secret.h, line: 34, column: 5
---CXXDestructor: ~Secret, type: [void ()] secret.h, line: 35, column: 5
---CXXMethod: foo1, type: [int (secret::Dim)] secret.h, line: 36, column: 9
---CXXMethod: foo3, type: [void (std::string)] secret.h, line: 37, column: 10
---CXXMethod: foo2, type: [void (secret::Dim, secret::Dim)] secret.h, line: 38, column: 17
---FunctionTemplate: foo4, type: [void (Variable<T, R>)] secret.h, line: 40, column: 14
----TemplateTypeParameter: T, type: [T] secret.h, line: 39, column: 24
----TemplateTypeParameter: R, type: [R] secret.h, line: 39, column: 36
---CXXMethod: getMessage, type: [std::string () const] secret.h, line: 41, column: 17
----NamespaceRef: std secret.h, line: 41, column: 5
----TypeRef: std::__1::string, type: [std::__1::string] secret.h, line: 41, column: 10
```

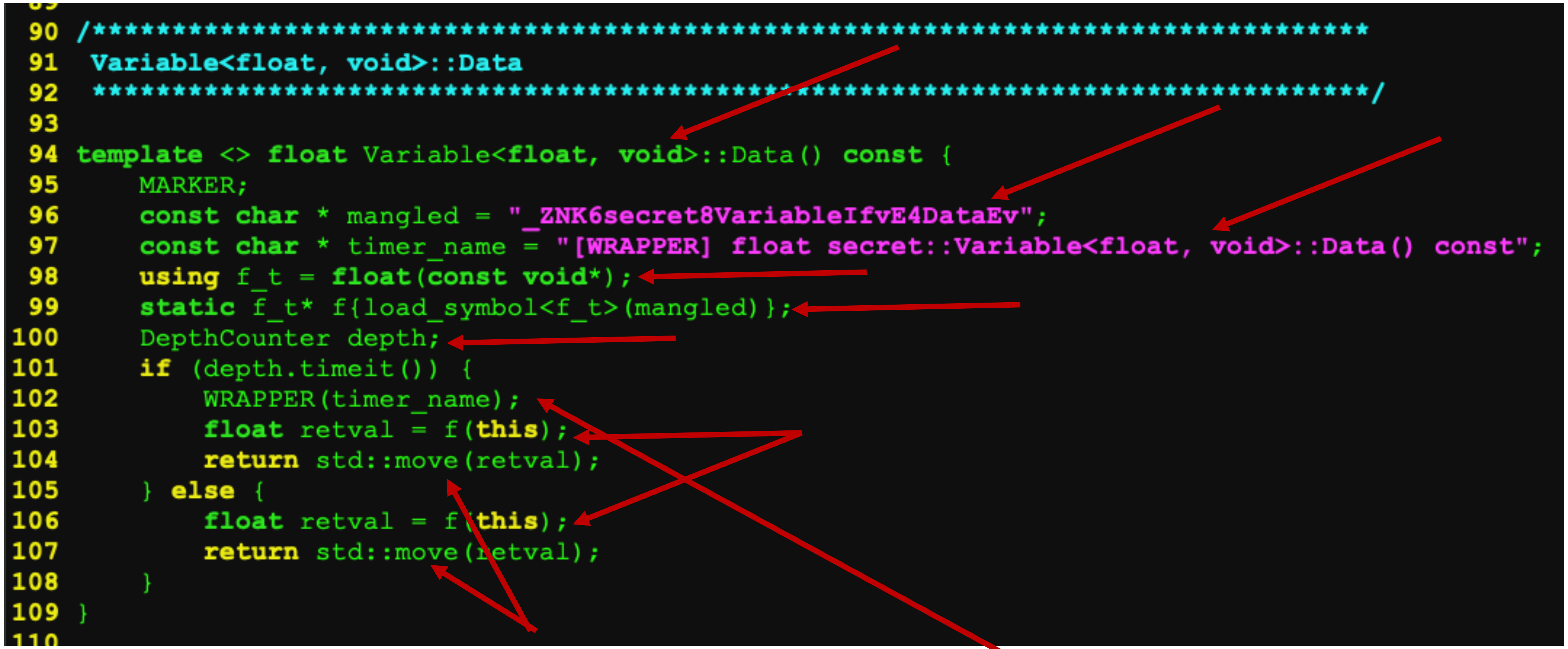
Helper code in wrapper

(fyi, ‘MARKER’ is just a printf debug helper)



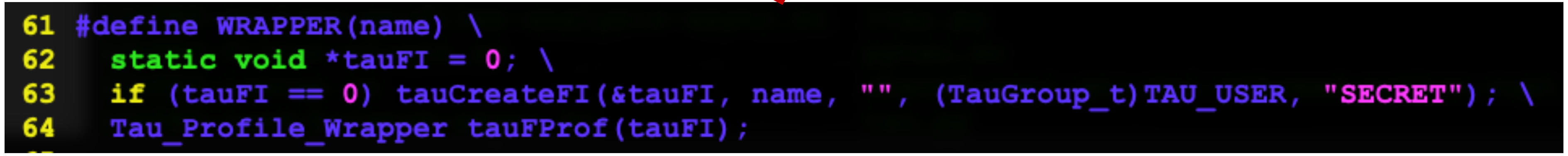
```
void * load_handle(void) {
    MARKER;
    const char * tau_orig_libname = "libsecret.so";
    static void *handle = (void *) dlopen(tau_orig_libname, RTLD_NOW);
    if (handle == NULL) {
        std::cerr << "Error opening library "
                  << tau_orig_libname
                  << " in dlopen call"
                  << std::endl;
    }
    return handle;
}

template<class T> T* load_symbol(char const* name) {
    MARKER;
    static auto handle = load_handle();
    void * tmp = dlsym(handle,name);
    if (tmp == NULL) {
        std::cerr << "Error obtaining symbol "
                  << name
                  << " from library"
                  << std::endl;
    }
    return reinterpret_cast<T*>(tmp);
}
```

Example: wrapped class template function



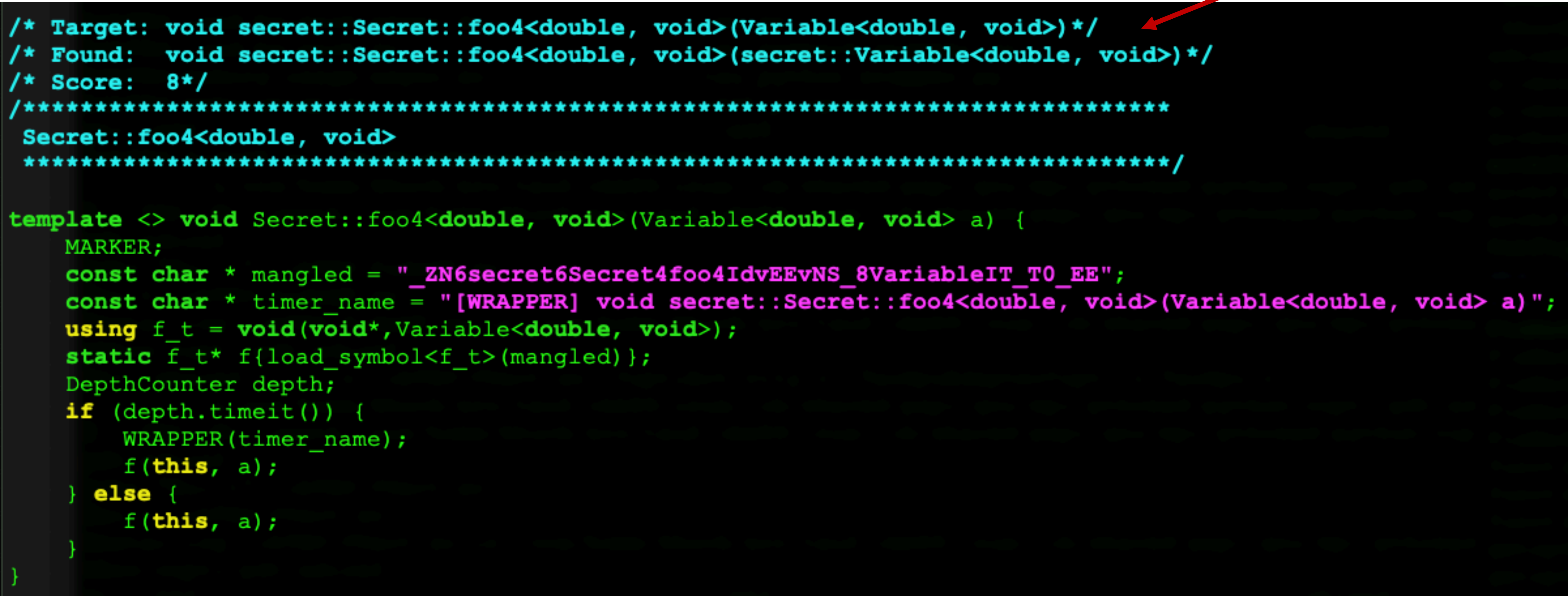
```
/***********************************************************************************
 Variable<float, void>::Data
 ***********************************************************************************/

template <> float Variable<float, void>::Data() const {
    MARKER;
    const char * mangled = "_ZNK6secret8VariableIfvE4DataEv";
    const char * timer_name = "[WRAPPER] float secret::Variable<float, void>::Data() const";
    using f_t = float(const void*);
    static f_t* f{load_symbol<f_t>(mangled)};
    DepthCounter depth;
    if (depth.timeit()) {
        WRAPPER(timer_name);
        float retval = f(this);
        return std::move(retval);
    } else {
        float retval = f(this);
        return std::move(retval);
    }
}
```

Can be any tool, not just TAU →



```
#define WRAPPER(name) \
  static void *tauFI = 0; \
  if (tauFI == 0) tauCreateFI(&tauFI, name, "", (TauGroup_t)TAU_USER, "SECRET"); \
  Tau_Profile_Wrapper tauFProf(tauFI);
```

Example wrapped template method

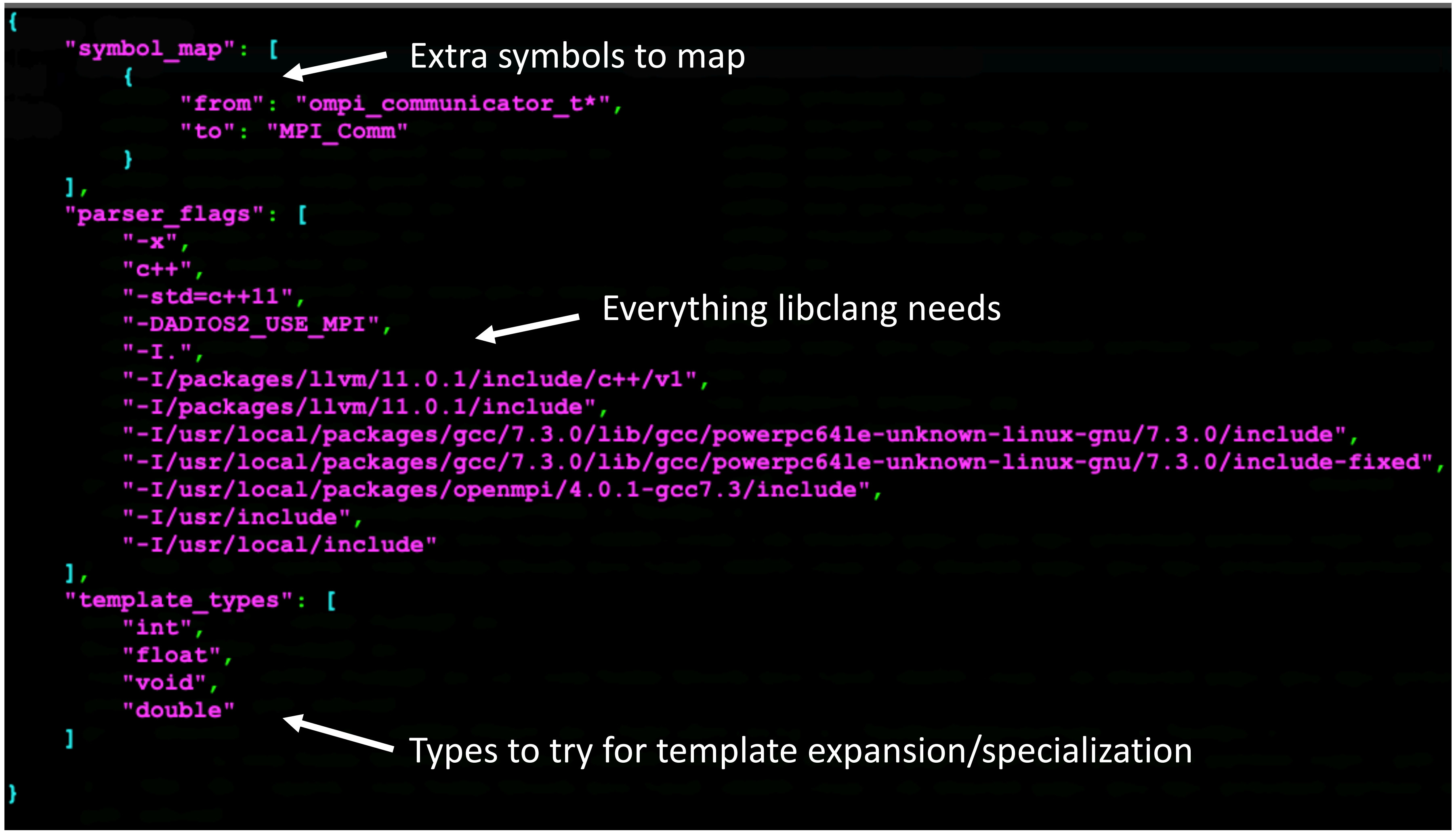


```
/* Target: void secret::Secret::foo4<double, void>(Variable<double, void>)*/
/* Found:  void secret::Secret::foo4<double, void>(secret::Variable<double, void>)*/
/* Score:  8*/
/*****************************************************************************
 Secret::foo4<double, void>
 *****************************************************************************/

template <> void Secret::foo4<double, void>(Variable<double, void> a) {
    MARKER;
    const char * mangled = "_ZN6secret6Secret4foo4IdvEEvNS_8VariableIT_T0_EE";
    const char * timer_name = "[WRAPPER] void secret::Secret::foo4<double, void>(Variable<double, void> a)";
    using f_t = void(void*,Variable<double, void>);
    static f_t* f{load_symbol<f_t>(mangled)};
    DepthCounter depth;
    if (depth.timeit()) {
        WRAPPER(timer_name);
        f(this, a);
    } else {
        f(this, a);
    }
}
```

Algorithm: https://www.geeksforgeeks.org/sequence-alignment-problem/ (slightly modified/debugged)

Config file

Types to try for template expansion/specialization (another example)

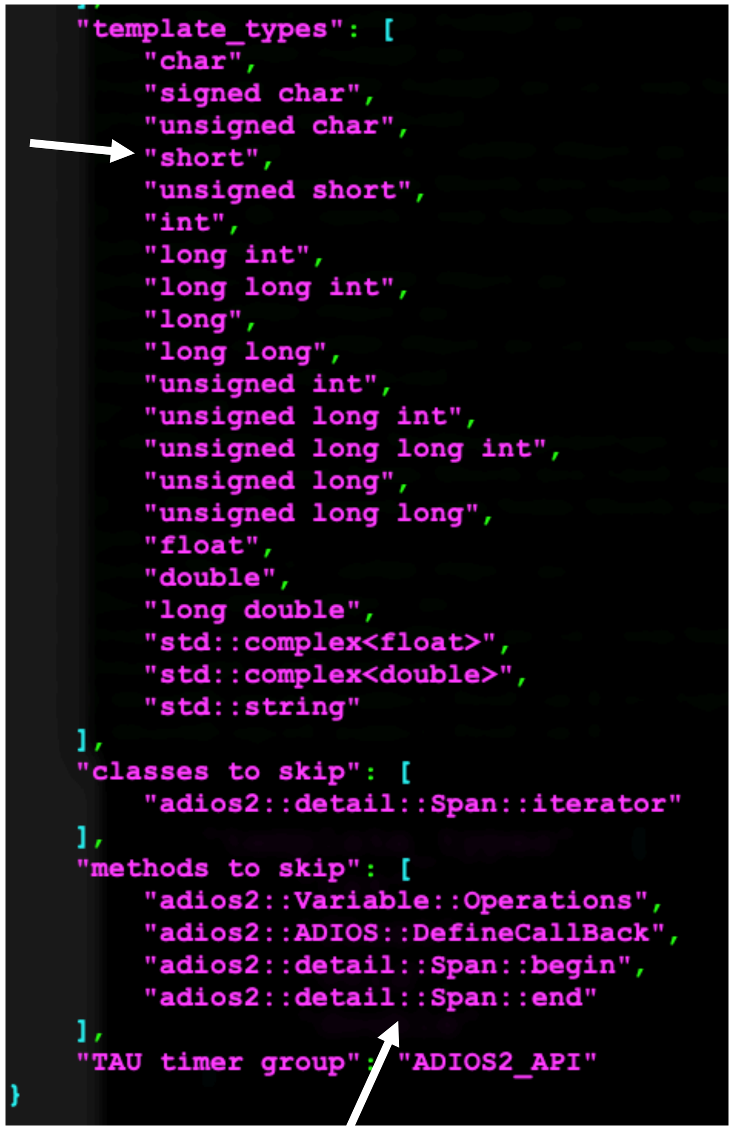


```
{
    "symbol_map": [
        {
            "from": "ompi_communicator_t*",
            "to": "MPI_Comm"
        }
    ],
    "parser_flags": [
        "-x",
        "c++",
        "-std=c++11",
        "-DADIOS2_USE_MPI",
        "-I.",
        "-I/packages/llvm/11.0.1/include/c++/v1",
        "-I/packages/llvm/11.0.1/include",
        "-I/usr/local/packages/gcc/7.3.0/lib/gcc/powerpc64le-unknown-linux-gnu/7.3.0/include",
        "-I/usr/local/packages/gcc/7.3.0/lib/gcc/powerpc64le-unknown-linux-gnu/7.3.0/include-fixed",
        "-I/usr/local/packages/openmpi/4.0.1-gcc7.3/include",
        "-I/usr/include",
        "-I/usr/local/include"
    ],
    "template_types": [
        "int",
        "float",
        "void",
        "double"
    ]
}
```

Extra symbols to map

Everything libclang needs

Types to try for template expansion/specialization



```
    "template_types": [
        "char",
        "signed char",
        "unsigned char",
        "short",
        "unsigned short",
        "int",
        "long int",
        "long long int",
        "long",
        "long long",
        "unsigned int",
        "unsigned long int",
        "unsigned long long int",
        "unsigned long",
        "unsigned long long",
        "float",
        "double",
        "long double",
        "std::complex<float>",
        "std::complex<double>",
        "std::string"
    ],
    "classes to skip": [
        "adios2::detail::Span::iterator"
    ],
    "methods to skip": [
        "adios2::Variable::Operations",
        "adios2::ADIOS::DefineCallBack",
        "adios2::detail::Span::begin",
        "adios2::detail::Span::end"
    ],
    "TAU timer group": "ADIOS2_API"
}
```

Selective wrapping, some TAU options

Final thoughts

- Not sure this is all that much more useful than Gotcha…
  - Demangled symbol (usually) has the full signature, but…
    - Missing return type sometimes (assume void?)
    - Missing const on some arguments (why?)
    - Member function…or just a function (do we need a “this” pointer?) (debatable…)
    - Distinguishing between namespaces and classes (does it matter? See above)
  - Gotcha is a little more manual, but that *could* be automated, too ☺
    - Cl[ia]ngWrap could automate the generation of the Gotcha code
- Cl[ia]ngWrap makes handling/parsing return types and arguments “easier” (but not easy)
- As long as string alignment matches correctly, all good (if not…boom!)