Linux Plumbers Conference | Richmond, VA | Nov. 13-15, 2023

Callsite Trampolines

Aleksei Vetrov (vvvvvv@google.com)
Steven Rostedt (rostedt@google.com)
Suren Baghdasaryan (surenb@google.com)

Motivation: Callsite tagging

Inject a structure and code to operate on that structure at the call site of the function.

- Codetag: generic structure to record a location in the source code (file, line, module)
- Created at compile time for each instrumented code location
- Placed into an array of codetags by the linker to iterate easily
- Gets embedded into an application-specific structure that records additional information, accessible using *container_of*
- Codetags from dynamically loaded modules are supported.

Current Applications

- Memory allocation profiling (slab, page, per-cpu, vmalloc).
- Dynamic fault injection
- Latency tracing
- Error source tracking

Framework can be easily utilized for more applications.

Latest version of Memory allocation profiling under review: https://lore.kernel.org/all/20231024134637.3120277-1-surenb@google.com/

Preprocessor macro approach (example application: memory allocation profiling)

```
#define alloc_hooks(_do_alloc, _res_type, _err)  \
({                                               \
  _res_type _res;                                \
  DEFINE_ALLOC_TAG(_alloc_tag, _old);            \
                                                 \
  _res = _do_alloc;                              \
  alloc_tag_restore(&_alloc_tag, _old);          \
  _res;                                          \
})
```

```
#define kmalloc_array_node(_n, _size, _flags, _node)   \
       alloc_hooks(_kmalloc_array_node(_n, _size, _flags, _node), void*, NULL)
```

Issues with the instrumentation

- Code pollution - lots of allocation functions wrapped into macros
- Name collisions

Feedback from upstream reviewers:

Provide simple way to mark functions to be instrumented and let tools do the instrumentation

Callsite trampolines proposal

Should inject both code and data structure

Injected data structure should be unique for each callsite

```
<kernel_func>:
[..]

call kmalloc

[..]

call kmalloc

[..]
```

```
<kmalloc>:

[..]

ret
```

```
<kmalloc>:

[..]

ret
```

Compiler support (working POC with CLANG)

```
typedef struct codetag {
    const char* file;
    int line;
    int counter;
} codetag;

__callsite_wrapper
static inline __callsite_ReturnT
wrapper(__callsite_Args args) {
    static codetag callsite_data =
          { __callsite_FILE, __callsite_LINE, 0 };
    callsite_data.counter++;
    return __callsite_Callee(args...);
}
```

```
__attribute__ ((callsite_wrapped_by(wrapper)))
int foo(int x) {
    return x;
}

int bar(const char* name, int x) {
    return foo(name, x);
}

int main(int argc, char** argv) {
    typeof(bar)* foo_ptr = &foo;
    foo_ptr(0);
    foo(1);
    bar(2);
    return 0;
}
```

Compiler implementation

`__callsite_wrapper` keyword transforms a function into a *C++ template* like this:

```
template<
  auto __callsite_Callee,
  typename ...__callsite_Args,
  typename __callsite_ReturnT=decltype(__callsite_Callee(__callsite_Args()...)),
  unsigned int __callsite_LINE=__builtin_LINE()>
static inline __callsite_ReturnT wrapper(__callsite_Args... args) {
  return __callsite_Callee(args...);
}
```

`__attribute__ ((callsite_wrapped_by(wrapper)))`

Transforms any mention of a function like `foo` into a `wrapper<foo>`

Compiler implementation

Advantages of using templates:

- Every callsite will have its own specialisation of the template
- One can take a pointer to this specialisation
  - Callsite wrapper is able to track calls through pointers
- Every specialisation has its own scope for function static variables

Disadvantages:

- Templates are not expected when clang is running in C mode
- Parameter pack expansion (...) doesn't exist in C mode

Compiler implementation

Transforming wrapper into a template:

- `__callsite_wrapper` is interpreted as "declaration specifier"
  - Mark the function declaration as a wrapper
  - Create template parameters
  - Add them to the declaration scope
- When a declarator is parsed as a function (`Sema::ActOnFunctionDeclarator`)
  - Create a `TemplateParameterList`
  - Enable C++ machinery to build a `FunctionTemplateDecl`

```
-  if (getLangOpts().CPlusPlus) {
+  if (getLangOpts().CPlusPlus || D.getDeclSpec().isCallsiteWrapperSpecified()) {
```

Compiler implementation

Wrapping functions:

- Add a new attribute `callsite_wrapped_by(wrapper)`
  - It stores the wrapper as an `Expr*`
- Add `WrappedByDecl` field to `LookupResult`
  - It will be filled when the wrapped function is looked up
- Use it to invoke the wrapper when `LookupResult` is used to create an expression
  - `Sema::ActOnIdExpression`, `Sema::ClassifyName`
  - The original function is passed as an explicit template parameter to the wrapper

Compiler implementation

Template instantiation:

- Instantiation is done using existing functionality for C++
- Parameter deduction
  - Callsite parameters are deduced from the callsite context
  - File and line are extracted from `SourceLocation`
  - `__callsite_Args...` is deduced automagically from arguments
  - `__callsite_ReturnT` has to be deduced manually

Compiler implementation

Name mangling:

- All template and static variables instantiations should have unique names
- Template arguments form part of the instantiation's mangled name
  - Example: `_Z7wrapperITnDaX3fooELm24ELA9_KcEJEEvv`
- Mangled name components:
  - Name of the wrapped function
  - Line number
  - **Filename length**
- Mangling has to be enabled explicitly in C mode for wrapped functions

Alternatives considered

Clang-tooling code transformation

- Use clang::annotate to mark wrappers and wrapped functions
- Generate intermediate source file with callsite variables filled in
- Advantages:
  - No changes to the compiler
  - Compatible with GCC
- Disadvantages:
  - Distribution of the tool: by source in the kernel tree?
  - Name mangling, wrapper instantiation, parameters deduction have to be reimplemented