CoogL

Concurrent
Object Oriented
Generic Language

COOGL

Pronounced
C-OOGL

www.COOGL.org

Ramon.Pantin@COOGL.org

Why COOGL?

- The worlds computer system infra-structure is mostly written in C or C++
- All of it is at risk from the “unsafe” nature of both languages
- COOGL is very close to C
- COOGL was designed to reengineer large C code bases, incrementally, and eventually make them “safe”

Why COOGL?

- Because C needs to evolve, C++ is too complex, both are unsafe
- Because we must do something about all of that unsafe code
- The kernels of most computer systems are written in C, they are all unsafe
- Windows, macOs/iOS, Android, ChromeOS, GNU/Linux, BSDs, AIX, Solaris, HP-UX, etc. are continuously at risk of being exploited

CoogL

What is it?

COOGL: What is it?

- Safe
- No undefined behavior
- Object Oriented
- Generic Programming
- Modules

COOGL: What is it?

- A simple language very close to C
- Compiled into readable C code
- Open source compiler
- Free to download book

CoogL

CLEAN, a
C subset

COOGL: CLEAN C subset

- CLEAN, pronounced “C-lean”
- Removed:
  - C preprocessor
  - Prototypes
  - K&R C style function declarations
  - Minor unimportant C quirks removed
- Code written in CLEAN can be shared between COOGL and C code

COOGL: CLEAN C subset

- Both CLEAN and COOGL have a context free grammar.
- *cast(type)expr* replaces: *(type)expr*
- When compiled as C code it will need to be included into a C file that has prototypes, other #includes, etc.
- Will need:
  - #define cast(type) (type)

COOGL: CLEAN C subset

- Proper subset, everything in CLEAN behaves the same in C and COOGL
- No silent changes, removed features cause compilation errors
- Minor differences with C:
  - Struct and union tags are proper types, not in a separate name space
  - Unions can not contain pointers, directly or indirectly
  - No variable argument functions

CoogL

Incrementally
Converting
a Kernel

COOGL: Converting a Kernel

- A kernel, say Linux, being reengineered into COOGL can be incrementally:
  - Turned into C+CLEAN code, all compiled with C compiler, slowly:
    - Initially 99.9% in C and 0.1% in CLEAN
    - Later 99% and 1%, then 95% and 5%, ...
  - The CLEAN parts can also be compiled as COOGL code in a parallel testing path
  - Eventually some parts compiled as COOGL
  - Rest compiled C+CLEAN

COOGL: Converting a Kernel

- Reengineered Incrementally:
  - Eventually whole kernel is COOGL
  - Then same process is followed for each kernel module (file systems, drivers, etc)
- Reengineering the kernel:
  - Start with kernel memory allocators
  - Kernel stack management
  - Remove pointers into and across stacks

CoogL

Design
Principles

COOGL: Design Principles

- Keep it as simple as possible
- Keep it as close as possible to C
- Learn from its ancestors (Simula67, Clu, and Eiffel)
- Be done with it, not a: “self balloning language” whose evolution never ends

COOGL: Design Principles

- Compiled into C so native C compilers that are used for kernel development can continue to be used
- No “shotgun marriage” into a compiler family (GCC or CLANG)
- Gain trust of C programmers by being able to “diff” source code and generated C code
- Preserve source code, indentation, comments, variable names, etc

CoogL

Unsafe
C code

Unsafe C code

```
void wrong() {
     char x[1];
     x[200] = 'x';
}
```

Unsafe C code

```
void store_at_100(int *p) {
    P[100] = 1;
}
void wrong() {
    int v = 1;
    store_at_100(&v);
}
```

Unsafe C code

```
char s[] = {"hi"};
char d[3];

void stomp() {
     s[2] = 'x';
     strcpy(d, s);
}
```

Unsafe C code

```
void smash() {
    char c;
    union {
        char *p;
        int i;
    } u;
    u.p = &c;
    u.i = 17;
    *u.p = 'x';
}
```

Unsafe C code

```
void store(char *p) { *p = 'x'; }

char *set(char *p) {
    *p = ' '; return p;
}
char *bad() {
    char c;
    return set(&c);
}
int main() {
    char *p = bad();
    store(p);
}
```

Unsafe C code

```
char *wrong() {
    char buf[6] = {"hello"};
    char *p = strchr(buf, 'e');
    return p;
}
```

- All of the unsafe C code examples above cause a compilation error when compiled as COOGL code

Unsafe C code

```
void freeit(char *p) {
    free(p);
}                          // different files
void stomp() {
    freeit(p);
    *p = 'x';
}
```

- Doesn’t cause a compilation error in COOGL, covered later in presentation

CoogL

Safety

COOGL: Safety goals

- Invalid memory accesses can not occur
  - What this means will be defined later
  - For now think: memory accesses than can be exploited by a hacker to take over the program's execution can not occur
- Preserve C's ability to carefully lay out data structures in memory, **safely!**

COOGL: Insight about safety

- Externalized memory:
  - Laid out to be sent over the network
  - Laid out to implement: databases, file system metadata, volume managers, file formats
  - Data placed on shared memory
- Externalized memory:
  - Does not contain pointers or pointer like data, such as array descriptors
  - Instead of pointers contains offsets or indexes

CoogL

Safety mechanisms

COOGL: Safety mechanisms

- Array descriptors
- Plain and non-plain data
- Restrictions on the use of class members that are plain data
- Global data can not point to local data
- No pointers between run-time stacks
- Stack growth is checked and probed
- Can not point to released stack memory

COOGL: Safety mechanisms

- Various cast operators to ensure safety
- NULL is unsafe, deprecated, use NIL
- Trapping addresses and pointer values
- Deconstructed values and uninitialized data
- Permanent association of heap virtual addresses and types

CoogL

Array descriptors

COOGL: Array Descriptors

- Pointer like, but describe an array of objects
- Array indexing is checked (and optimized)
- Arguments declared with [], [][], [][][], etc are array descriptors
- Implemented by the language:
  - Atomically fetched and stored
  - Can only be indexed when they are on the stack (local variables or arguments)

COOGL: Array Descriptors

```
char *strcat(char dst[],
             char src[]) { ... }
```

- Source code compatible between C and COOGL
- In C it means that **dst** and **src** are both pointers to char

COOGL: Array Descriptors

- Array descriptors have these compiler managed members, they can not be changed individually:
  - **start**: points to first element of array
  - **end**: points one past the last element of array
  - **max[N]**: number of elements in each of N dimensions
- Array walking with array descriptors is always valid
- Invalid pointer value computation not allowed
- Out of bounds indexing raises an exception
- Optional dummy objects at **start-1** and **end**

COOGL: Array Descriptors

- Unidimensional array descriptors implemented with two words:
  - **start** and **max[0]**
  - **end** computed by the compiler, if needed
- Allows atomic fetch and store of unidimensional array descriptors to be very fast
- Modern architectures support double-word atomic load and store operations
- Atomic fetch and store of multidimensional array descriptors is more complex

CoogL

Plain and
non-plain data

COOGL: Plain and non-plain data

- Plain data:
  - Data that **does not contain** pointers, array descriptors, variable length arrays, or variables of index or uindex type
  - Objects of user defined classes **are not plain data,** even if they do not contain none of the above.
- Non plain data:
  - Data **that contains** pointers, array descriptors, variable length arrays, or index/uindex variables.
  - Objects of user defined classes

COOGL: Deconstructed values

- Deconstructed values, for example after destruction, are:
  - NIL for all the pointers
  - All the array descriptors have dimensions set to zero, and begin and end set to NIL
  - All other data is set to zero

COOGL: Uninitialized data

- Logically is set to the deconstructed values prior to construction
- But this is not required because the constructor or copy constructor would fail compilation if they used uninitialized data (it must first be constructed or set)

CoogL

“Branded” virtual addresses

COOGL: “Branded” virtual addresses

- Heap memory has its pages branded to the type of objects that they contain
- Memory is reused at the physical level, not the virtual level
- Linux mremap() and userfaultfd() mechanisms aid in the physical memory reuse and the incarnation of deconstructed memory
- **preclass**, reuse counts, and virtual dispatch

CoogL

cast operators

COOGL: cast()

```
cast(type) value

cast(type *) addr
```

- **cast** non-pointer value, same as C
- **cast** with pointer value, allowed if type is plain data and size of **type** is same or smaller than size of object that **addr** refers to
- Otherwise it results in a compilation error

COOGL: up_cast()

```
up_cast(type *, value) ptr
```

- Assume that **type** is a derived type that inherits from the type that the static type of **ptr** points to, directly or indirectly. Otherwise the up_cast() results in a compilation error.
- If at run-time ptr is actually points to an object whose type is type (or a type that derives from it), then the up_cast() succeeds an it results in a pointer with the value of **ptr** but of type: **type** *, otherwise the result has the value **value** usually NIL, or a trapping address, or the address of an object type **type**
-

COOGL: uptr_cast()

```
uptr_cast(type *, value) uptrval
```

- If **uptrval** is a trapping pointer value, then the result is a pointer to **type** with **uptrval** value
- Otherwise its a pointer to **type** with value **value**, usually **NIL**, or another trapping pointer value, or the address of an object of type **type**

COOGL: try_cast()

```
try_cast(type *, mem, value) addr
```

- If **type** is plain data, and if **addr** considered as a pointer to **type**, would result in a pointer to data within the **mem** array descriptor (which much describe plain data):
  - then the cast succeeds and its value is the same the same the C cast: **(type *) addr**
- Otherwise:
  - the cast fails and its value is **value** which must be either **NIL**, a trapping address, or the address of data of type **type**

COOGL: try_cast()

```
struct header  { uint h1, h2; };
struct prefix  { uint p1, p2; };
struct body    { uint b1, b2; };
struct postfix { uint pfx1; };
struct footer  { uint f1; };
```

- Pretend those are part of a protocol that has a header, an optional prefix, a body, an optional postfix, and a footer.

COOGL: try_cast()

```
void make_message(uint data[],
            header *h, prefix *pre,
            body *b, postfix *post,
            footer *f)
{
    uchar *ptr = cast(uchar *) data;
    *try_cast(header *, data, NIL)
        ptr = *h;
    ptr += sizeof(header);
    ...
```

COOGL: try_cast()

```
...
if (pre) {
    *try_cast(prefix *,
              data, NIL)
        ptr = *pre;
    ptr += sizeof(prefix);
}
...
```

COOGL: try_cast()

```
...
if (b) {
    *try_cast(body *, data, NIL)
        ptr = *b;
    ptr += sizeof(body);
}
...
```

COOGL: try_cast()

```
    ...
    if (post) {
        *try_cast(postfix *,
                  data, NIL)
            ptr = *post;
        ptr += sizeof(postfix);
    }
    *try_cast(footer *,
              data, NIL) ptr = *f;
}
```

```
void make_message(uint data[], header *h, prefix *pre,
                  body *b, postfix *post, footer *f) {
    size_t size = sizeof(header) + sizeof(footer);
    if (pre) size += sizeof(prefix);
    if (b) size += sizeof(body);
    if (post) size += sizeof(postfix);
    lang__COND_STORE(data.max[0] < size, NIL, 0); // one < test
     //  lang__COND_STORE() is a compiler and hardware barrier,
     //  stores below only issued if no exception was raised
    uchar *ptr = (uchar *) data;
    *(header *) ptr = *h;
    ptr += sizeof(header);
    if (pre) {
        *(prefix *) ptr = *pre;
        ptr += sizeof(prefix);
    }
    if (b) {
        *(body *) ptr = *b;
        ptr += sizeof(body);
    }
    if (post) {
        *(postfix *) ptr = *post;
        ptr += sizeof(postfix);
    }
    *(footer *) ptr = *f;
}
```