Low-level Debugging

Improving your Source Code

Optimizing a Case Away

In GCC, there's a built-in function called `__builtin_unreachable()`. Your code should never call this function. The purpose of this is to mark a certain condition in your code as something that will never happen, so the compiler can optimize it away.

> **ASIDE:** Upcoming version **C23** has the function `unreachable()` which means the exact same, and this can be used in any compiler compliant with the spec.

```
int sqrt(int n) {
  // Suppose you know by precondition n is non-negative
  if (n < 0)
    __builtin_unreachable();
  /* code */
}
```

This is a message from you, the developer, to the compiler telling it to never worry about the case of `n` being negative. This unlocks some optimization techniques, like something like `n >> 3` for division, which wouldn't work for negative numbers. Of course, if you don't ensure the precondition or abuse `__builtin_unreachable()`, you could cause undefined behavior. **The compiler trusts that you know what you're doing.**

Notice how there's no need for conditional branching that could slow down the code. You use an `if` statement, but the compiler optimizes it away because of what it means. No code gets generated for that block.

You could also transform this into a macro:

```
#define assume(x) (x ? 0 : __builtin_unreachable())
```

Maximizing Cache Efficiency

A technique for fitting variables into *cache lines* to maximize computations per cycle:

```
// This means allocate x at a multiple of 16
// Arg to aligned() must be a power of 2
long x __attribute__((aligned(16)));
```

You can also mark functions based on how often they are likely to be called at runtime:

```
// This function likely won't be called
int f(int) __attribute__((cold));
// The function likely WILL be called
int g(int) __attribute__((hot));
```

We want to divide code into two regions, "hot" and "cold". We want to make the hot region small, consisting of code we will execute a lot. Cold code can be as big as we want. The main point is that the code in the hot region will be *cached*. Cold code would sit in DRAM. This makes efficient use of our **instruction cache**.

Improving your Binary with GCC Options

Security Improvements

Attempt to protect against **stack overflow attacks**:

```
gcc -fstack-protector
```

This generates extra code in your function. It is as if you added something like:

```
int f() {
  // added check
  int i = randomvalue; // some macro probably

  char buf[512];
  read(fd, buf, 512);
  x = buf[12];

  // added check
  if (i != randomvalue)
    abort(); // reliably crash, dump core

  return x;
}
```

This means that attackers would have to guess the random value, meaning they'd have to guess from the possible values `int` can take on, and even more if something like `long` is used.

This works because the attacker would have to overwrite the `randomvalue` to get to the bottom of the call frame to reach the return address, which is typically what they need to overwrite to continue their attack.

There is debate whether `-fstack-protector` is a default option. Some Linux distros like Debian and Ubuntu have it on by default. This is off by default on SEASnet. There's also the opposite option called `-fno-stack-protector` that turns the setting off.

> **ASIDE:** The **halting problem** states that there is no algorithm for looking at a program and figuring out if it would halt. There's no way to look at a program to figure out if it would stack overflow (Alan

> Turing).

Performance Improvements

The `-O`, `-O2`, `-O3`, etc. flags to specify the level of *optimization* ***for CPU usage***.

```
gcc -O foo.c
```

This is off by default because it slows down the compiler. The compiler spends more time finding ways to optimize the machine code to make the *runtime performance* faster. It's a trade off between compile time and runtime performance.

Additionally, the higher the level of optimization, the harder it becomes to debug the machine code because the compiler may choose to take certain liberties for the sake of making the executable more performant, making the machine code less parallel with the original source code.

By default, it cares about runtime performance, not executable size. To *optimize for size*, you use the `-Os` flag.

There is another optimization option that is still under construction `-Og`. This means to *optimize for debugging* i.e. make the code as "debuggable" as possible without affecting performance.

The `-O0` flag specifies to *not optimize at all*.

ASIDE: Consider this case where turning optimization on breaks the code:

```
double *p;
*--p = *p++ * *p++;
// "pop pop multiply push"
```

If you have competing side effects in the same statement, this is *undefined behavior*. The compiler is allowed to perform the actions in whatever order.

The compiler "activated" the bug by relying on undefined behavior in the name of optimization.

Automating Hot/Cold with Profiling

1. You compile the program with a special flag that inserts extra code to count the number of times each instruction is executed. This understandably slows down execution.
2. You then run the program to gather the statistics.
3. Then, you recompile the program *with* the statistics. That way, the compiler knows which instructions are hot and cold and can decided how to mark them itself.

As long as your test runs are representative of how your program will actually be run, this is a great way of improving the performance of your application.

Other GCC options

```
gcc --coverage
gprof
```

Inlining

```
gcc -flto
```

Normally, GCC optimizes one module at a time.

```
// m.c
static int f(int x) {
  return -x;
}
```

Then suppose in the same module you call:

```
// m.c
f(3);
```

The compiler can deduce at compile-time that `f(3)` will always resolve to `-3`, so it can just expand it to `-3`. This is called **inlining**.

Link-Time Optimization

Then suppose you have definition and call in different modules:

```
// m.c
int f(int x) {
  return -x;
}
```

```
// n.c
extern int f(int x);
```

```
// n.c
f(3);
```

By default, the compiler can no longer make this expansion. It instead has to go through the overhead of a normal function call: putting `$3` into a register, etc.

The point of `gcc -flto` is to instruct the compiler to perform the optimization *across module boundaries* - whole program optimization.

```
gcc -flto foo.c bar.c baz.c
```

This creates a `foo.o`, `bar.o`, etc. that contains machine code, but within the file is also a copy of the source code. Thus, by the time you link them together:

```
gcc -flto foo.o bar.o baz.o
```

GCC has a copy of the entire source code, so it can perform the inline optimization to generate better code. The *downside* is that this command takes a long time. The optimization algorithms that GCC uses are around $O(N^2)$. It could take literal days with a large program.

Catching Bugs with Static Checking

Debugging your program *before* the program runs. "Static" here has nothing to do with static allocation/static lifetime; it just means when the program isn't running. This is the opposite of **dynamic checking**.

- ➕ No runtime overhead.
- ➕ Covered bugs are 100% prevented by the time the program runs.
- ➖ Some bugs can not be statically checked for 100% reliably.

Example of static checking:

```
static_assert(0 <= n);
```

The implication however is that the expression e.g. `0 <= n` MUST be computable at compile-time. If it isn't, it's a compiler error.

EXAMPLE: Suppose you have a program that assumes a `long` is 64 bits:

```
#include <limits.h>
static_assert(LONG_MAX >> 31 >> 31 == 1);
```

This is once again like a *message from you to the compiler*. The compiler then checks beforehand for this condition before attempting to compile the program.

GCC Warning Flags

In general, the flags beginning with `W` mean "warning". `-Wall` means "turn on all warnings":

```
gcc -Wall
```

However, people found that this tends to output extraneous/unnecessary warnings. Nowadays, `-Wall` has come to mean "turn on all warnings that most people will find useful."

`-Wall` implies:

- `-Wcomment`: warn about valid but bad comments like `/* a /* bad comment */`
- `-Wparentheses`: warn about style considered to be bad because something probably forgot parentheses in situations where operator precedence may be less familiar, like in `i << j + k` or `i < j || k < l && m < n`. The latter is common knowledge, but GCC disagrees, so this flag can be controversial.
- `-Waddress`: warn about making pointer comparisons when you probably didn't mean to, like `p = strchr("ab", "b"); if (p == "b") ...`
- `-Wstrict-aliasing`: warn about trying to "cheat" with pointers.

  ```
  int i;
  long *p = (long *)&i;
  ```

  The C/C++ standard states that this results in undefined behavior, even if `int` and `long` happen to be the same size. This is controversial because a lot of low-level programs like the Linux kernel uses this all the time, *very carefully*. This is also why **casting** in general is risky because optimizing compilers may not do what you expect.

- `-Wmaybe-uninitialized`: warn about possible paths through a function where you use an uninitialized variable.

  ```
  int f(int i)
  {
    int n;
    if (i < 0)
      n = -i;
    return n; // if i >= 0, uninitialized n is returned
  }
  ```

  If you do something that checks out with arithmetic reasoning, the compiler may or may not be find with it, depending on other flags you pass it.

  ```
  int f(int i)
  {
  ```

```
    int n;
    if (i < 100)
      n = -1;
    if(i <= -1)
      return n;
  }
```

Thus, there could be false positives if the compiler is not smart enough to deduce that the combination of things you've done always returns an initialized variable.

GCC Extra Warning Flags

`-Wextra` is a collection of warnings like `-Wall` that are more controversial/less useful.

Some flags it includes are:

`-Wtype-limits`: warn about comparisons where the answer is obvious due to the type.

```
unsigned u;
if (u < 0)
  /* This code would never run! */
```

This is not always trivial. For example, if you're trying to run portable code, some `typedef`s might be different:

```
#include <time.h>
time_t t;  // could be signed or unsigned
if (t < 0)
  /* This code may or may not run! */
```

In recent versions of GCC:

```
gcc -fanalyzer
```

This turns on **interprocedural warnings**. This looks at all callers and callees of a function to come up with a better picture of, say, if a variable is uninitialized or not. In other words, it looks through all *all paths through all calls* of every function (in current .c file, as that is what is deducible at compile-time). This is disabled by default because it slows the compiler down.

Specifying both of these flags would theoretically be rewarding but very expensive:

```
gcc -fanalyzer -flto
```

Helping the Compiler through Source Code

You can modify your source code in minor ways to help the compiler do better checking.

_Noreturn

```
// Prototype for some function that will never return
_Noreturn void fatal_error(char const *);

size_t size_sum(size_t a, size_t b)
{
  if (a <= SIZE_MAX - b)
    return a + b;
  fatal_error("size overflow");
}
```

Without the special `_Noreturn` signature, GCC will complain because it thinks size_sum is not always returning `size_t` even though it always does because `fatal_error` exits the program.

This will be standardized in C23 as `[[noreturn]]`.

`_Noreturn` also lets the compiler check whether a function declared to not return actually doesn't:

```
void fatal_error(char const *x)
{
  puts(x);
  exit(1);
}
```

`exit` is a `<stdlib.h>` function declared as something like:

```
_Noreturn void exit(int);
```

So `fatal_error` as it is defined above is fine. If we were to leave off `exit(1)`, then GCC would know that something is wrong.

Pure Functions

With GCC, you can declare a **pure function** with the signature, telling the compiler that this function should not modify the state of the machine:

```
int hash(char *buf, size_t bufsize) __attribute__ ((pure));
```

This is stronger than merely declaring the parameters as `const`. A pointer to `const` just promises to not modify an object via its pointer, not that its a pointer to an actual constant.

This way, if you do something like:

```
char buf[100];
char c = buf[0];
i = hash(buf, sizeof buf);
// c == buf[0] must be true
// This gives the compiler opportunities to cache in register, etc.
```

This will be standardized in C23 as `[[reproducible]]`;

Const Functions

Not to be confused with the `const` qualifier on C++ methods.

```
int square(int) __attribute__ ((const));
```

This is an even stronger flavor of pure functions. The return value can not even depend on the current state of the function, only its arguments.

A benefit is that if you call a `const` function multiple times with the same arguments, it could optimize knowing they would return the same value.

This will be standardized in C23 as `[[unsequenced]]`.

Multiple Attributes

Example:

```
// ... is a syntax feature for variable number of arguments
int my_printf(char const *fmt, ...) __attribute__
  ((nonnull(1) | format(printf, 1, 2)));
  // nonnull(1) means the first argument cannot be NULL
  // The format attribute does some type checking inside fmt arg:
  // my_printf("a=%s", 1024); // not okay!
```

The names inside the `__attribute__` label, like `nonnull` and `format`, are not actual functions. They are *keywords* to the `__attribute__` syntax.

Catching Bugs with Dynamic (Runtime) Checking

An example of dynamic checking, where you check at *runtime* and abort or similar if something fails:

```
#include <assert.h>
int f() {
  /* code */
```

```
  assert(0 <= n);
}
```

Checks you put in yourself. This is very flexible because it's your own code.

```
#include <stdckint.h> // C23 only
if (ckd_add(r, a, b))
  return EOVERFLOW;
// Not sure what this is but yeah
```

GCC -fsanitize

Alternatively, you can let the compiler *insert* runtime checking.

```
gcc -fsanitize=undefined
```

Generate extra code to make the program reliably crash if it attempts undefined behavior (except for addresses). This causes the program to be slightly slower.

At the machine level, the adding overflow may be implemented like:

```
addl %eax,%ebx
jo   ouch       ; jump if overflow

ouch:
    call abort
```

This is the same as `-fsanitize=undefined`, but for addressing errors, such as subscripting an array. Due to technical reasons, this actually only catches *most* addressing errors, not all.

```
gcc -fsanitize=addresss
```

This attempts to insert runtime checks for **memory leaks**. Memory leaks are technically not undefined behavior nor errors.

```
gcc -fsanitize=leak
```

This attempts to insert runtime checks for **race conditions**, where there may be multiple threads that attempt to access the same I/O resource.

```
```

```
gcc -fsanitize=thread
```

Most of the time, these flags are useful for development only. They could be left in for production if the application prioritizes safety over performance.

Valgrind

You can run a program called Valgrind on any program, no special compilation flags:

```
valgrind emacs
```

This runs the program in a special environment and allows for more runtime checking.

The summary screen displays information about memory leaks.

- ➕ No special flags needed when compiling.
- ➖ Much slower. Checking is also less extensive because Valgrind only has access to the machine code and not the underlying source code.

BIG POINT: Runtime checking is dicey. Even with all these tools, bugs can still slip through.

> **ASIDE:** You can manually raise a compiler error with preprocessor directives in C:
>
> ```
> #if INT_SIZE == LONG_SIZE
>  #error "ouch"
> #endif
> ```

Portability Checking

Portability is a large part of software construction issues.

Portability Concerns

- **Architecture**: 32 vs. 64-bit platforms?

```
gcc       # default is 64-bit
gcc -m32  # check for 32-bit
```

- **OS:** Linux vs. MacOS vs. Androids vs. ...?
- **Software:** Chromium vs. Frefox vs. ...? (JS)
- **Software version:** Chromium v102 vs. v107 vs ...?

You need a good strategy to tackle this. One strategy is to have a comprehensive testing system. But you also need to consider the source code. The way it's done is defining a **portability layer**, a level of abstraction:

```
main code
    |
(some API)
    |
portability module
    |
Chromium, etc.
```

> The compiler should be your servant, not your master. If you get warnings, look at what they're actually saying. If they're false positive, shut them off. **- Dr. Eggert, probably**

Debugging Strategies

1. You're better off NOT debugging at all. Often times projects are stuck in **debugging hell**, spending more time debugging than actually developing. Debugging is an inefficient way of finding and fixing bugs. If you're spending a lot of time debugging, you should probably change your software construction approach such that you don't get so many bugs. *Be proactive* e.g. use static checking!

2. Write test cases. **Test-driven development (TDD)** is a theory of development that states that test cases are higher priority than code; one should write test cases first, the idea being you can use the test cases to debug the specifications of the code before actually writing the code.

3. Use a better platform.

4. **Defensive programming**. "When you're driving, assume that everyone is an idiot, drunk or both." Assume the other modules are broken.

    1. You can use traces and logs (`print()` statements!).

    2. Checkpoint/restart. Have one primitive in your program that saves the state of your entire application, probably to some file where it persists. Then have another primitive that restores the state:

        ```
        save_state("foo")  # do this periodically
        restore_state("foo")
        ```

    3. Assertions: crash when something that should never happen occurs.

    4. Exception handling.

    5. **Barricades**.

        ```
        +----------------+----------------+
        |    possibly    |   clean data   |
        |    bad code    |    structs     |
        +----------------+----------------+
                         ^
        ```

```
    some barricade that processes
    code from the outide world, which
    you assume to be bad, before it
    makes it into your code
```

6. **Interpreters** (such as Valgrind) that execute code in a special environment.
7. **Virtual machines** that run a program in its own special sandbox, isolated from the outside system.

> **TDD ASIDE:** If you write some test cases, and your program passes all the test cases, then you screwed up because you haven't found the bug you wanted to find. When you're writing test cases, you're trying to be imperfect. You're trying to think "how do I make this program crash?" Often times, tests are written by another class of developers because the self-interest of coders causes them to write bad test cases. **- Dr. Eggert, probably**

Debugging Tools

Some examples are **GDB**, the GNU debugger, and **Valgrind**.

strace is a command that outputs the system calls a program uses, as if there were many `printf` logging calls in the source code.

```
#      vvvvvvvvvvvvvvvvvvv any shell command
strace cat /etc/os-release
```

Other commands we've seen before like `ps` and `top` are also *DevOps tools*. System administrators use this to monitor server activity, but they can also be useful for debugging.

How Debuggers Work

GDB is actually a separate process from the process being debugged. It uses special system calls to exert some control over the process being debugged. Model:

```
(gdb process)--+
               | special system calls
               v
        (your process)
```

These special system calls include, at any point in execution:

- Starting/stopping/continuing
- Accessing memory
- Accessing registers

These do have security restrictions and cannot be used arbitrarily. These restrictions depend on the OS, but typically the rule is that the debugger must control a process with *same user ID*. Other OS may have a more

restrictive rule, stating it can only debug a *child process*.

Getting Started with Debugging: GCC

You can use a `-g` flag to specify debug info level for a C program:

```
gcc -g3 program.c
```

This conflicts with optimization because code that is optimized by the compiler tends to become harder to understand.

```
gcc -g3 -O2 program.c
```

This typically results in **inlining**, where calls to functions can be optimized away by substituting their body into places where it was called. The produced machine code would then be functionally equivalent but have lost the information that a function was even called.

What the `-g` flag does is bloat the resulting object and executable files with debugging tables. This data isn't visible when the program runs normally, but debuggers will be able to access them.

> **ASIDE:** `_FORTIFY_SOURCE` is a standard technique used by GCC to make stack overflows less likely to succeed. For technical reasons, this is incompatible with no optimization i.e. attempting `gcc -O0` will cause a compiler error.

Starting GDB

Dropping a program into GDB:

```
#   vvvv program to debug
gdb diff
```

Setting the working directory of the program when it starts up:

```
(gdb) set cwd /etc
```

Setting environment variables for the debugging session:

```
(gdb) set env TZ America/New_York
```

A defense technique against buffer overflow attacks is to have the program run at randomized locations in memory (CS 33). By default, Linux executes programs in an environment with randomized addresses for the

stack, heap, C library, etc. and many even the `main()` function.

The downside of this program is that it will run differently every time. This means that if there's a bug that depends on stack addresses for example, then it may appear sometimes and not for others. This makes debugging harder, so by default, this option is already on:

```
(gdb) set disable_randomization on
```

Actually running the program. The arguments you supply after `run` are in shell syntax and forwarded to the executable being debugged:

```
(gdb) run -u /etc/os-release - < /dev/null
```

Alternatively, you can make GDB be in charge of another program using the PID of running process, effectively suspending it.

```
(gdb) attach 986317
```

Releasing the program:

```
(gdb) detach
```

Controlling Your Program

`^C` stops the program. GDB takes control.

`*(int *)0=27` crashes the program and falls under GDB's control.

Continue running the code:

```
(gdb) # c
(gdb) continue
```

Single step through the source code. Similarly, single step through the machine code.

```
(gdb) # s
(gdb) step
(gdb) # si
(gdb) stepi
```

Stepping can be tricky because there isn't always a sequential mapping of source code lines to machine code lines. Stepping through some machine code lines may make it look like the program is jumping back and forth between source code lines instead of running one-by-one in order.

A courser-grained variant of the `step` command. Advancing to the next line of source code at the current function call level i.e. a single step but without worrying about function calls, stepping "over" them. Similarly, it has a machine code version.

All these commands control the **instruction pointer**, e.g. `%rip` on x86-64.

```
(gdb) # n
(gdb) next
(gdb) # ni
(gdb) nexti
```

Finish the current function and then stop:

```
(gdb) fin
```

You can use **breakpoints** to stop the program at a certain instruction, typically a function name. Creating a breakpoint:

```
(gdb) # b
(gdb) break analyze
(gdb) break diff.c:19
```

Listing your current breakpoints and their numbers:

```
(gdb) info break
```

Deleting a breakpoint by number:

```
(gdb) del 19
```

You can use **watchpoints** to tell the program to run until the specified variable `p` changes value:

```
(gdb) watch p
```

How does GDB implement breakpoints?

GDB takes the process being debugged and modifies its machine code. It stomps on the machine code of the specified function/line/instruction by zapping the first byte with a special instruction that is guaranteed to cause the program to trap, allowing GDB to take control.

How does GDB implement watchpoints?

Single step through the code, and after each instruction, see if `p` has changed. This can be really slow unless youh have special hardware support for watchpoints. Many CPUs, including x86-64, have this support.

Other GDB Commands

Printing a C expression (or register values):

```
(gdb) # p
(gdb) print expr
(gdb) print $rax
(gdb) print a[5]
(gdb) print cos(3.0)
```

It does more than just allow you to look at data. It lets you run a subroutine like `cos` in the program, which can modify the data and/or call arbitrary code from other parts of the program.

Disassembling a function to get the assembly code:

```
(gdb) disas cos
```

You can set a **checkpoint** and then run the code from the checkpoint by its number:

```
(gdb) checkpoint
(gdb) restart 42
```

The inverse of `continue` is `reverse-continue` (`rc`). This means to start running the program backwards until it hits the most recent breakpoint that it passed. This tends to be *very expensive* because GDB has to set a bunch of checkpoints under the hood.

```
(gdb) # rc
(gdb) reverse-continue
```

For **cross-debugging**, you can specify what target you want to run in

```
(gdb) target
```

This makes GDB run on some virtual machine or something?