Applying C/C++ *sanitizers* on the Nix toolchain

Farid Zakaria fmzakari@ucsc.edu

November 2021

1 Introduction

As hardware has steadily increased in power (computational, storage etc..), software has become increasingly complex to match. Using an audit of various programming languages and their dependencies as a proxy, shows that the mean number of transitive dependencies can be on the order of 10-50 dependents[1]. If you account for the dependencies necessary to build (i.e. compiler) as well as runtime dependencies, it's easy to see that software and their relationships can be viewed as a dense graph. Often times these relationships are implicit, Nix aims to make it explicit.

Nix is a purely-functional package manager which can be configured via a language of the same name. Nix was published in a PhD thesis by Eelco Dolstra in 2006[1]. It focuses on letting users specify repeatable actions on files, which make it possible to write portable, reproducible actions in Nix.

The Nix community is incredibly vibrant. The *nixpkgs*[2] repository, which contains the various build recipes has close to 4000 unique contributors and is currently the largest and *freshest* (most up to date) package repository.

Given the popularity and use of Nix, what is the code quality of the underlying toolchain? We present data after having applied a memory *sanitizer* (dynamic code analysis) and report on the defects found and the commentary on modifications necessary to the codebase to support it.

2 Sanitizers

Programmers are fallible, and the result of their mistakes are bugs. These bugs can be on account of a misunderstanding of the algorithm attempted

[1]`https://edolstra.github.io/pubs/phd-thesis.pdf`
[2]`https://github.com/NixOS/nixpkgs/`

to be implemented, the language specification or from more mundane typos. There has been a resurgence in tooling and languages that aim to help programmers from either making certain class of mistakes or catching them earlier during the software development process[3].

C/C++ does not trade performance for safety, and famously is known to *make it easy to shoot yourself in the foot*[4]. Sanitizers are a suite of tools, open-sourced by Google, that can be *statically* included into the built binary to help catch a large suite of errors such as data races, memory leaks or use of uninitialized memory[3]. Other tools exist similarly but are prohibitively expensive as they could impose a cost of 100x slowdown. The various sanitizers are shown to provide high fidelity information necessary to track and fix the bug while incurring only a modest 2.5x slowdown in execution time.

Address Sanitizer A detector of various memory use errors such as buffer overflows and memory leaks.

Thread Sanitizer A data race detector, a data race occurs when two threads access the same variable concurrently and at least one of the accesses is write.

Memory Sanitizer A detector for the use of uninitialized memory.

3 Evaluation

3.1 Building with Clang

In order to augment the build system to include the sanitizers, a deeper understanding of Nix's build system is necessary. There is a limited *Hacking Guide* at `https://nixos.org/manual/nix/unstable/contributing/hacking.html`, which at the time of this report details how to manually kick off the Autotools[5] commands to `configure && make`.

Running the build commands within the *nix-shell*[6] on a build machine with 256GiB memory and 128 cores (AMD Ryzen Threadripper 3990X 64-Core Processor) but limiting build to use 28 sub-processes takes close to a minute (51s).

[3] `https://clang.llvm.org/diagnostics.html`

[4] `https://www.stroustrup.com/quotes.html`

[5] `https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html`

[6] a reproducible environment offered by Nix that it uses to build itself

```
[nix-shell]$ ./bootstrap.sh
[nix-shell]$ ./configure $configureFlags \
                 --prefix=$(pwd)/outputs/out
[nix-shell]$ make -j28
```

By default, the Nix code-base relies on GCC to build itself, specifically version 10.3.0. The repository use to be able to build itself with either *clang* or *gcc*, however that toggle functionality looks to have been removed. There are unmerged contributions from the community which seek to reintroduce this capability[7]. The Nix codebase already has support for changing the standard environment (*stdenv*) to one that uses clang instead (*clangStdenv*) and support can be reintroduced to the codebase with a minimal patch[8].

Fortunately as the codebase use to build with Clang, *at some point*, the build remained compatible and no subsequent changes were necessary to build with clang. Although GCC has support for building with the various sanitizers, Clang was chosen since the sanitizers reside within the LLVM repository, and therefore there is the assumption for Clang to have superior support.

3.2 Building with Address Sanitizer

The build was augmented to include support for Address Sanitizer (ASan), following the official documentation of the LLVM website[9]. The key change to introduce to the build step is to compile and link the program with `-fsanitize=address`, and making sure to link using *clang* and not *ld*. Additional changes were made to have the tool emit nicer stack traces and remove incompatible linker flags such as `-Wl,-z,defs` (report unresolved symbols in object files)[10]. Unfamiliarity with the codebase and Autotools added complexity when discovering all the places to inject additional linker flags (`LDFLAGS`) as failure to do so resulted in cryptic unresolved reference errors.

```
error: undefined reference to '__asan_after_dynamic_init'
src/nix/log.cc:56: error: undefined reference to '__asan_set_shadow_f5'
clang-7: error: linker command failed with exit code 1 (use -v to see invocation)
make: *** [mk/lib.mk:118: src/nix/nix] Error 1
```

[7]https://github.com/NixOS/nix/issues/4129
[8]https://gist.github.com/fzakaria/00dafaec3b3f0864d136470bf6579099
[9]https://clang.llvm.org/docs/AddressSanitizer.html
[10]https://gist.github.com/fzakaria/eab8d14695549ddefdf9ba09038016ad

3.3 Discovering Errors

Running the Nix binary surprisingly immediately emits many failures caught by the LeakSanitizer (LSan) and the AdressSanitizer (ASan). The LeakSanitizer is bundled automatically by building with the AdressSanitizer. Running the Nix binary, innocuously to simply emit it's version information, results in a memory leak.

```
./src/nix/nix --version
nix (Nix) 2.4

=================================================================
==1172266==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 152 byte(s) in 1 object(s) allocated from:
    #0 0x553ac6 in calloc (/nix/src/nix/nix)
    #1 0x7fa2229b0b92 in __pthread_attr_extension (libc.so.6)

Direct leak of 16 byte(s) in 1 object(s) allocated from:
    #0 0x5876ef in operator new(unsigned long) (/nix/src/nix/nix)
    #1 0x7fa222e0b5d5 in nix::makeSimpleLogger(bool) logging.cc:130:12
    #2 0x7fa222e0b5d5 in __cxx_global_var_init.6 logging.cc:26
    #3 0x7fa222e0b5d5 in _GLOBAL__sub_I_logging.cc logging.cc

Indirect leak of 32 byte(s) in 1 object(s) allocated from:
    #0 0x553d2e in realloc (/nix/src/nix/nix)
    #1 0x7fa2229b0d13 in pthread_attr_setaffinity_np@GLIBC_2.3.4 (libc.so.6)

SUMMARY: AddressSanitizer: 200 byte(s) leaked in 3 allocation(s).
```

Upon investigation, this memory leak is created at startup from overriding a default global *logger* pointer variable with other object addresses, without freeing the memory beforehand. The Nix codebase is built against the C++11 standard which includes support for *smart pointers* (i.e. `std::unique_ptr`), however the logging variable's lifetime is managed manually. An upstream pull-request[11] has been made to remove the memory leak.

Unfortunately after resolving the above memory leak, subsequent runs of the Nix binary results in *many more*, specifically involving the AST & parser. The code maintainers explain that the AST & parser knowingly

[11] https://github.com/NixOS/nix/pull/5579

cause many memory leaks since tracking them is unecessary overhead and they are generally needed for the lifetime of the application[12].

```
$ nix search jdk
==1526543==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 5432 byte(s) in 97 object(s) allocated from:
    #0 0x7139cf in operator new(unsigned long)
    #1 0x7f644b4f1e13 in nix::EvalState::addPrimOp(nix::PrimOp&&) eval.cc:637
    #2 0x7f644b6ad46b in nix::EvalState::createBaseEnv() primops.cc:3719
    ...
    ......
    ............
SUMMARY: AddressSanitizer: 14982 byte(s) leaked in 331 allocation(s)
```

The sanitizer also caught a variety of memory bugs that are all different variants of *use after free*.

```
$ nix build -f concurrent.nix
==2023391==ERROR: AddressSanitizer: heap-use-after-free on address 0x606000d3d4ad at
pc 0x00000067893e bp 0x7ffd73767570 sp 0x7ffd73766d20
READ of size 6 at 0x606000d3d4ad thread T0
    #0 0x67893d in __interceptor_memcpy.part.0 (nix)
    ...
```

The exploration of the binary for errors was done *ad-hoc* since there were so many discovered failures rather than building a working corpus of tests. A future improvement would be to run the full test-suite of the toolchain itself with a debug build that includes ASan.

3.4 Results

The errors discovered by the sanitizer were easy to locate as the binary still contained meaningful debug symbols and steps were taken following the prescribed setup to disable certain compiler optimizations that may obfuscate the location. Familiarity with the C/C++ language was necessary in order to rectify the problems as the sanitizer only locates the failure but does not provide any advice how to resolve it. At times the bugs were trivial

[12] https://matrix.to/#/!VRULIdgoKmKPzJZzjj:nixos.org/$RczAfZSEWi7kXO4NYHSdm2q3hzf1iS63CNEGSc6FEpc?via=nixos.org&via=matrix.org&via=nixos.dev

to identify, however some *stack-use-after-free* with *string_view* required a more nuance understanding of the language to solve.

```
string create_str() {
    return "Hello World!";
}

string_view crop_exclamation(string_view str) {
    return str.substr(0, str.size() - 1);
}

int main() {
    // BOOM! hello_world here is invalid memory
    string_view hello_world = crop_exclamation(create_str());
    std::cout << hello_world << std::endl;
    return 0;
}
```

Listing 1: Demonstrating a suble stack-use-after-free failure with string_view

Three upstream contributions were made to fix errors reported by the sanitizer, two of which have already been merged[13][14]. The contributions were welcomed and have even inspired further investigation into the codebase via the sanitizers by other active members of the community[15].

4 Compiler Design and Implementation

The sanitizes are instrumented into the code at compile-time rather than than dynamically at runtime, which other similar tools have done (i.e. valgrind). The address sanitizer specifically overrides *free* and *malloc*, to catch memory failures (memory leaks or use-after-free) that the compiler is unable to detect itself during compilation. Performing the instrumentation at compile-time results in a less severe speedup penalty[2]. Additionally, the sanitizers are tune-able, changing various settings that for instance may control corrupted heap memory size to stack unwinding depth. These settings can effect the ability to discover bugs or the amount of diagnostic infor-

[13] https://github.com/NixOS/nix/pull/5599
[14] https://github.com/NixOS/nix/pull/5592
[15] https://github.com/NixOS/nix/pull/5599#issuecomment-976683952

mation presented to the developer both of which may affect the speedup penalty imposed.

5 Conclusion

The suite of sanitizers available to statically instrument into a codebase is a surprisingly easy and effective method to discover a wide class of failures that are often challenging to reason through or discover simply by reading the code. The sanitizers impose a small enough penalty to the performance of the application that applying them to complex software systems such as Chromium is doable and worthwhile.

The *address sanitizer* was added to the Nix toolchain and a suite of errors was easily uncovered. The diagnostic information was complete enough to identify the bugs, fix them and upstream contributions have been merged. A surprising amount of bugs were uncovered through the exercise yet do not meaningfully effect the functionality of the tool. An interesting question for future work is a deeper discussion on what constitutes a bug if it doesn't typically impose a failure.

References

[1] Riivo Kikas et al. "Structure and Evolution of Package Dependency Networks". In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 2017, pp. 102–112. DOI: `10.1109/MSR.2017.55`.

[2] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX ATC'12. Boston, MA: USENIX Association, 2012, p. 28.

[3] Evgeniy Stepanov and Konstantin Serebryany. "MemorySanitizer: fast detector of uninitialized memory use in C++". In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 46–55.