Final Paper

Firmware Module for Infer Static Analyzer

Yousef Elsendiony (YE2914) and Tony Butcher (ab4359)

Introduction

In this project, we looked to develop a module for Infer targeted at finding firmware related failures in C language that the Infer documentation acknowledged it did not check for. Some of these common failures include stack overflows, null dereferencing, memory leaks, unwanted compiler optimizations, and cast exceptions. The midterm research paper done by Yousef included more details regarding why these are common failures in firmware. Our goal was to address the following three research questions:

1. *Do other static analyzers cover bugs that Infer acknowledges it doesn't check for? If so, can we leverage those results towards building new modules for Infer?*

2. *How practical is it to add a new static analysis module to Infer?*

3. *Is there functionality we can add that can improve Infer, specifically in areas where the documentation acknowledges it doesn't check?*

Our intent was to begin by comparing the results from Infer runs against other static analyzers for the C language. A better or more effective analyzer should find a higher number of real bugs found and a lesser number of false bugs or noise. The criterion would be to introduce sample bugs similar to infer and demonstrate how the modules we build can detect these sample firmware failures. Another criterion for rating the effectiveness of our static analyzer would be to run against multiple firmware projects and examine the catch rate of errors.

We primarily examined the NVM express (NVME) command line interface tool. Our static analyses examined the existing code, as well as injected code based on identified misuse of the `volatile` C declaration, as noted by Professor John Regehr at the University of Utah.[1] Two misuses of `volatile` were added into one of the C files in the NVME package, argconfig.c, listed as examples 5 and 9 on his website.

[1] Regher, John. "Nine ways to break your systems code using volatile." *Embedded in Academia, Feb 26, 2010.* https://blog.regehr.org/archives/28

Setup and Repositories

Building Infer with the clang compile option was done in a VM, Ubuntu 16.04LTS with 12GB RAM and 80GB disk, and took about 90 minutes. Using less RAM or disk space when compiling clang resulted in failure.

Installing Infer:

```
$ sudo add-apt-repository ppa:avsm/ppa
$ sudo apt-get update
$ sudo apt upgrade
$ sudo apt install git libgmp-dev libsqlite3-dev zlib1g-dev libmpfr-dev
libmpfr-doc cmake m4 automake autotools-dev libcap-dev opam pkg-config
gcc
$ cd Desktop
$ git clone https://github.com/facebook/infer.git
$ cd infer
$ sudo apt install clang
$ ./build-infer.sh clang
$ sudo make install
```

Installing NVME (project files used to test infer):

```
$ cd ~/Desktop
$ git clone https://github.com/linux-nvme/nvme-cli
$ cd nvme-cli
$ infer run -- make
```

Installing bmcWeb (examined, but not included in final project):

```
$ cd ~/Desktop
$ git clone https://github.com/openbmc/bmcweb
$ mkdir build
$ cd build
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=1 ..
$ cd ..
$ infer run --compilation-database build/compile_commands.json
```

Our repositories:

- https://github.com/YousefElsendiony/infer/ - Infer project with custom modules (substitute into the git clone line of *Installing Infer* to use)
- https://github.com/YousefElsendiony/nvme-cli - NVME with modified test code in argconfig.c file (substitute into the git clone line of *Installing NVME) to use*

Location of previous project assignments and other static analysis results:

- https://github.com/YousefElsendiony/infer/tree/master/Papers
- https://github.com/YousefElsendiony/infer/tree/master/staticAnalyserResults

Research Questions

1. Do other static analyzers cover bugs that Infer acknowledges it doesn't check for? If so, can we leverage those results towards building new modules for Infer?

Static analysis of NVME was done using Flawfinder, cppcheck, and RATS. cppcheck yielded no results. Flawfinder and RATS yielded very generic results based on the presence of vulnerability prone C functions such as `memcpy()` and `system()`, although RATS did identify one more specific potential race condition issue:

```
415  nvme.c:1593: Medium: stat
416  A potential TOCTOU (Time Of Check, Time Of Use) vulnerability exists.  This is
417  the first line where a check has occured.
418  The following line(s) contain uses that may match up with this check:
419  1425 (scandir)
```

Our hypothesis after is that NVME, being a relatively mature project, has likely gone through static analysis checks already by its developers and had its more serious issues addressed.

After identifying the `volatile` declaration as a candidate for additional testing to find cases of types of bugs Infer could not identify, we added two misuses of it into the NVME argconfic.c file:

```
55  volatile int ready;
56  int message_Tony[100];
57  void foo_Tony (int i_Tony) {
58    message_Tony[i_Tony/10] = 42;
59    ready = 1;
60    printf("%d", ready);
61  }
62
63  volatile int ready2;
64  void foo_tony2(void){
65    ready2 = ready2;
66  }
```

After running the other analyzers again, we found their results were unchanged. Ultimately, we found we were unable to leverage purported functionality that these static analyzers claimed to have which Infer did not, and then decided to build modules based on additional bug research.

2. How practical is it to add a new static analysis module to Infer?

First, any changes to Infer require it be recompiled. Although the initial compile of clang took about 90 minutes for the initial installation, we found that incremental builds took about three minutes.

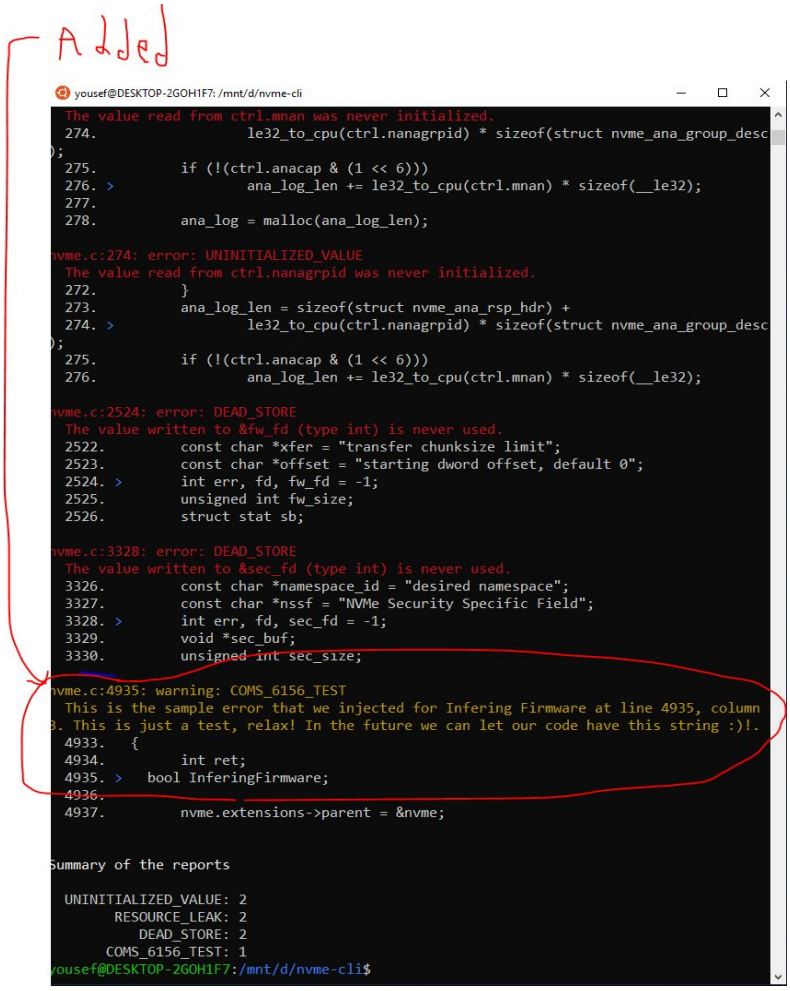
The practicality of adding the modules themselves depends on the complexity of the check being done. As a simple test case, we were able to add a bug check for the existence of a particularly named variable by making modifications to the Infer /infer/lib/linter_rules/linters.al file:

```
164 DEFINE-CHECKER COMS_6156_TEST = {
165
166   LET name_contains_test = declaration_has_name(REGEXP("TonyTiger"));
167
168   SET report_when =
169             WHEN
170                                name_contains_test
171                         HOLDS-IN-NODE VarDecl;
172
173   SET message = "This is the sample error that we injected for Infering Firmware Tony";
174   SET suggestion = "Tony's test :)!";
175
176 };
```

We then made the respective addition to the argconfig.c file of NVME:

```
49  /*start COMS 6156 test code*/
50  static int tonytiger;
```

After we recompiled and ran Infer with the changes, it was able to successfully identify our findings as shown in the screen capture below:

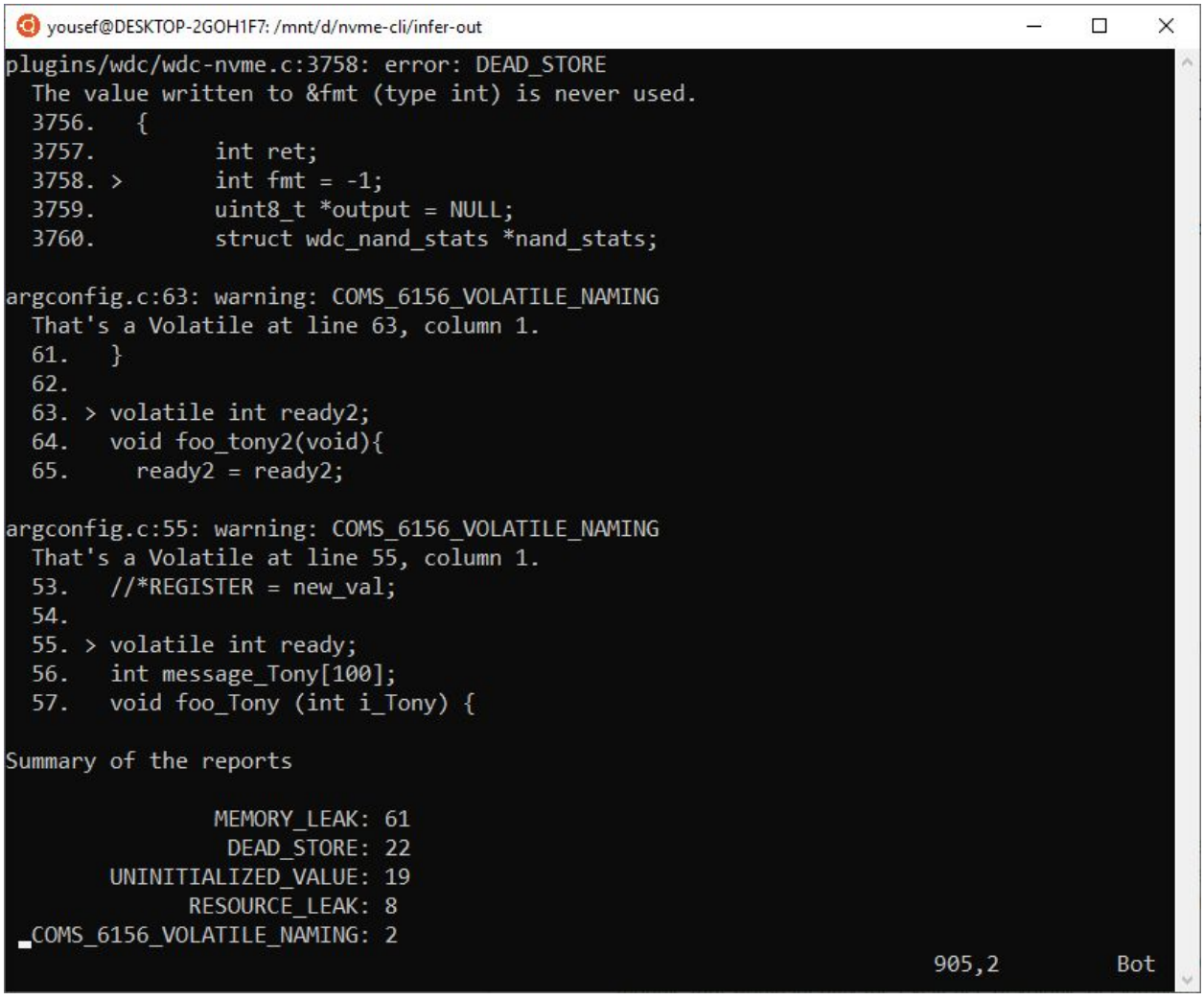


Adding a check for `volatile` was more complex and required the addition of a new predicate in the /infer/src/clang/cPredicates.mli file, which is addressed in greater detail in research question 3.

3. Is there functionality we can add that can improve Infer, specifically in areas where the documentation acknowledges it doesn't check?

We felt gaining a thorough understanding of misuse of the volatile declaration would require significant study outside of the scope of this class. We decided the best next step would be to add a module that would identify the existence of volatile declarations, and give notice to the users about its potential problems similar to the generic advice given by Flawfinder and RATS.

Understanding the misuse of the volatile declaration is complicated because the usual problems are that variables are not volatile, or volatile is overused to the point where developers are unsure which variables actually need to be volatile and which could be optimized by compiler. Making a checker requires a formula which can be run on every node of the abstract syntax tree. We decided the best next step would be to add a module that would identify the existence of volatile declarations, and give notice to the users about its potential problems similar to the generic advice given by Flawfinder and RATS:



The `volatile` check is seen above as type COMS_6156_VOLATILE_NAMING for users to easily find all instances of volatile, and then re-evaluate whether the volatile qualifier is appropriate.

What We Learned

Yousef: I learned quite a bit regarding the OCaml Language and how static analyzers worked. It took quite some time to get Infer working ; however, it proved worth it because I can now use the tool in my work life and show others how to use it quickly. Also now I have an base understanding of how differential/ separational logic works and an application to that in a static analyzer. I also would like to thank Facebook for their AL (abstract syntax tree) language which made it definitely much easier to use than hacking directly on the source code to make checkers for our static analyzer. Eventually I had to still add API's we needed to make a newer checker regarding volatile, so I learned a bit of where code is and how segmented lexing, parsing, and analysis is done in Infer to construct the AST to analyze. This semester was a long journey, but skills across the board were used which was refreshing and fun to have learned about a tool I can use outside of the course. Also to add, the most valuable skill I got from the semester was becoming better at working on open source projects and dealing with dependencies when building from source. I found several dependencies which was tough to deal and I usually haven't had to do more than running the ./config or setup.sh, so I now feel better equipped to debug future build problems when working with open source linux projects.

Tony: I've previously been a consumer of static analysis results but this is my first time exploring the effort that goes into building one as well as considering how compilers work. Unlike Yousef, I have not had exposure to Columbia's PLT class to better understand compilers and I don't have the expertise in the use of the C declarations we were researching to fully understand their potential misuse cases. Nevertheless, I gained an appreciation of how advanced Infer is in its checks compared to the other free static analyzers we examined, the checks that clang does against C code (as it refused to compile a couple of the misuse cases of volatile which I tried) and how complex building a good static analyzer can be, as many of the results from the other analyzers were cursory, and adding a module more complex than simply checking for the presence of a variable name required some intimate knowledge of compiler workflow.

Appendix

Static analyzers examined (Tony):

Static analyzer	**Last update**	**Opportunities**	**Results / Comments**
Flawfinder	2005	uses of risky functions, buffer overflow (strcpy()), format string ([v][f]printf()), race conditions (access(), chown(), and mktemp()), shell metacharacters (exec()), and poor random numbers (random()).	Results for Nvme and bmcweb, seem tailored around the generic use of specific C functions such as sprintf and strncpy rather than deep analysis of how the code is constructed, and the use of staticly sized arrays
Cppcheck	Feb 2010	invalid STL usage, overlapping data in sprintf, division by zero, null pointer dereference, unused struct member, passing parameter by value	Tests in nvme and bmcweb yielded no results. Hypothesis is since there are relatively mature projects that the developers conducted static analysis and resolved any findings which would have been found by cppcheck.
RATS	Sep 2013	Buffer overflows, race conditions	Nvme: buffer overflow; time of check/time of use findings Bmcweb: mostly static sized arrays similar to flawfinder; one finding for src/kvm_websocket_test.cpp: 105: Medium: read "Check buffer boundaries if calling this function in a loop and make sure you are not in danger of writing past the allocated space."
UNO https://gith	Mar 2019	Uninitialized variables,	Could not install with make, documentation does

ub.com/nimble-code/Uno		null-pointers, and out-of-bounds array indexing	not discuss installation. Submitted issue to gitweb page and hope to receive feedback given the last update was recent