Tool Operational Verification and Validation Results Framework for CodeHawk C Analyzer (Tool B)

1 Introduction

This document is part of a research Case Study simulating a Formal Methods Tool qualification exercise under DO-330, for the CodeHawk C Analyzer static analysis tool. An actual qualification exercise would produce, among other documents, a Tool Operational Verification and Validation Results document describing the qualification test results this for analyzer, as specified by DO-330 10.3.4. Because this is a research study, in which there is no actual qualifying organization and accompanying context, and because the tool under consideration is a research implementation without specific versions, release control, user documentation, or standard configurations, we will be describing a hypothetical set of results that might be the content of an actual Results document. This document is thus a Framework for an actual version of such a document, organized according to the DO-330 enumeration of required contents in section 10.3.4. It is based on a particular set of test case results that were previously produced to fulfill a DHS research contract for the tool's development.

2 Results Framework

[From DO330-10.3.4] **The tool operational verification and validation process produces the Tool Operational Verification and Validation Results, which should:**

A. For each review, analysis, and test, indicate each procedure that passed or failed during the activities and the final pass/fail results.

Discussion

As outlined in the *Tool Operational Verification and Validation Cases and Procedures Framework for CodeHawk C Analyzer* document, the test cases chosen for this simulated study do not cross foot with the individual requirements for C memory safety enumerated in the TOR and TR documents. Those requirements are defined over the space of statement variety for the C language. Any single C program will contain many of these statement types, and not contain many others. Since we have chosen a test suite consisting of seven large C applications, for reasons outlined in *Cases and Procedures* document, each of the seven test runs contains many individual memory constructs upon which a qualifying analyzer's output could be deemed to pass or fail. There is no summary judgment per test case.

If a qualifying organization had chosen instead to use input cases such as the Juliet Test Suite, which consists of small, toy programs each containing one vulnerable memory

construct, the individuation of the test cases would map one-one with the construct-specific memory safety requirements.

With the machine-readable metrics from the Gold Standard benchmark over the seven C applications we have chosen to use for this study, it would be possible to account for a qualifying analyzer's pass/fail status on each known vulnerability. If the analyzer reports the vulnerability, it passes. If not, it fails. This would be a measure of the tool's completeness. Conversely, each vulnerability that the analyzer reports that is not on the canonical list would count as a failure of soundness. A total result of no false positives would constitute (defeasible) evidence of soundness.

B. Identify the configuration item or tool version reviewed, analyzed, or tested.

Discussion

This is a simulated qualification, referencing a research implementation of the analyzer, so there are no actual tool versions to specify. The research version of the tool was built and tested on various versions of MacOS and Linux. In a real qualification, there would be a specific qualifying version to cite here.

C. Include the results of tests, reviews, and analyses.

The following results are taken from the actual testing/verification that was done during the DHS Gold Standard project to create an objective vulnerability reference for the 8 C test applications. Regarded as a DO-330 testing qualification exercise, it would have been rather circular, since the results of the CodeHawk C Analyzer (and the evaluation of these results by it's developers) were taken as canonical. Confidence in these results can now be used to objectively qualify other static analyzers. This situation exhibits a general problem for formal methods analyzers that take computer programs as input. The soundness challenge for such tools is in producing correct results for the (infinite) combinatorial complexity of programming language constructs. Small, toy programs will not probe this complexity. Real-world applications will, but the correct results will not be known in advance. Human analysts must comb through each reported vulnerability, and each safety assertion, to verify that all of the results are correct.

For an example of what a test evaluator would see in the CodHawk output, and how that relates to the original source code locations, the reader is referred to Appendix A of the *Theoretical Soundness Issues Report* produced as deliverable 2 for this project. (The example is not from the Gold Standard test suite). Below, we will summarize the results from the entire 8-application Gold Standard test suite.

CodeHawk places *primary* proof obligation of memory safety at every program location where an undefined memory reference is possible according to the C language semantics. It then uses abstract interpretation to generate invariants at each such location that will serve as the premises for a memory safety proof. Many of these obligations will not be able to be discharged from the invariants alone. When this occurs, the additional invariants needed to discharge the proof are assumed, and placed as pre and post

conditions on the containing function, or on referenced global data structures. These lifted obligations must be satisfied by all callers of the containing function and all modifiers (and creators) of the referenced data structures. We call these lifted conditions *secondary* proof obligations. A new round of (abstract) invariant generation will then be performed to provide lemmas for these secondary obligations. Undischarged secondary obligation will, in turn, be lifted to new secondary obligations on their containing functions and data structures. This iterative process continues until no new secondary obligations are generated.

Figure 1 below summarizes the number of primary proof obligations originally placed in each of the test applications (see *Tool Operational Verification and Validation Cases and Procedures Framework for CodeHawk C Analyzer, section B*), color-sorted by memory safety category (see *Tool Operational Requirements (TOR) Framework for CodeHawk C Analyzer*, section E.1).

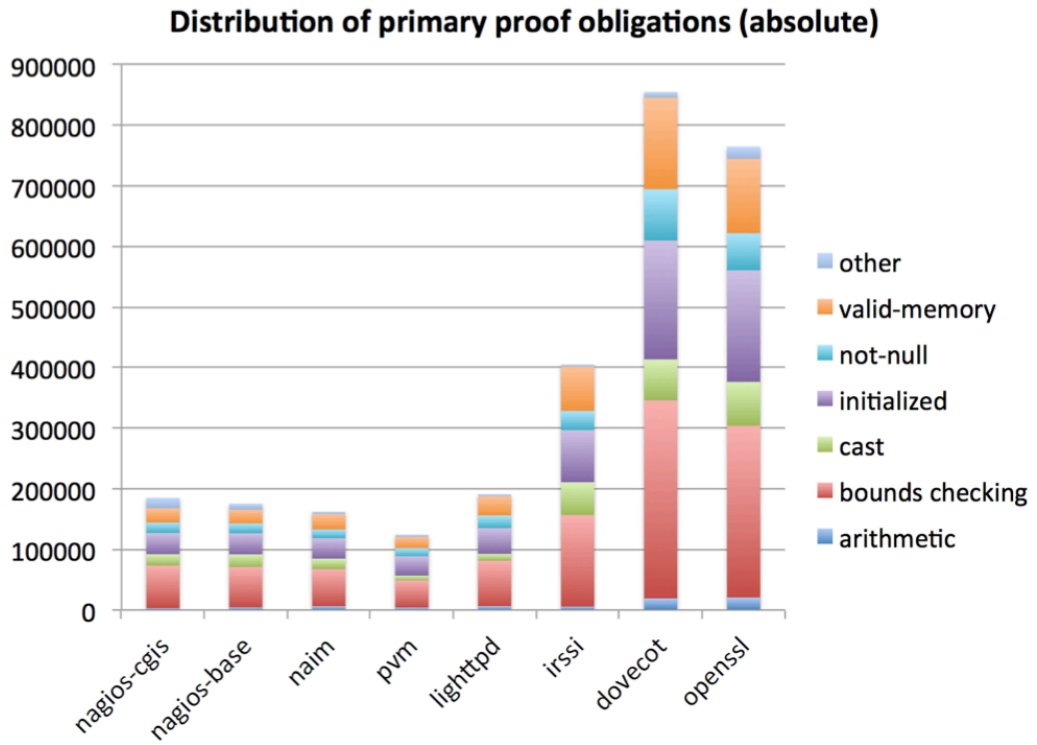
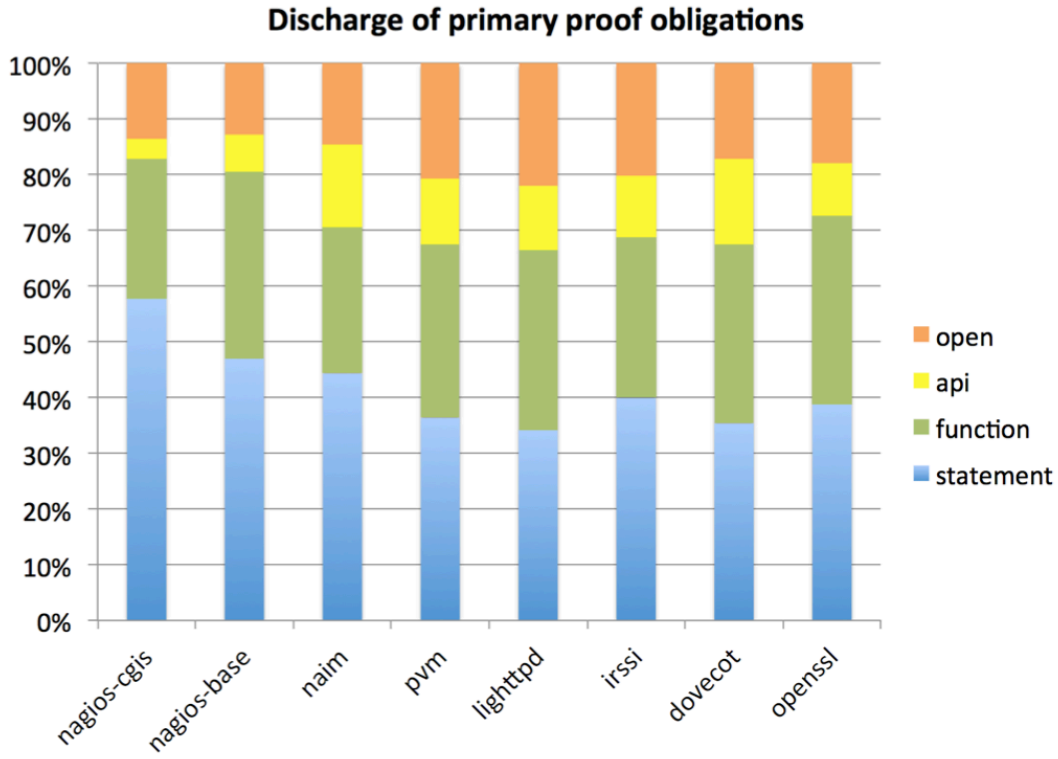
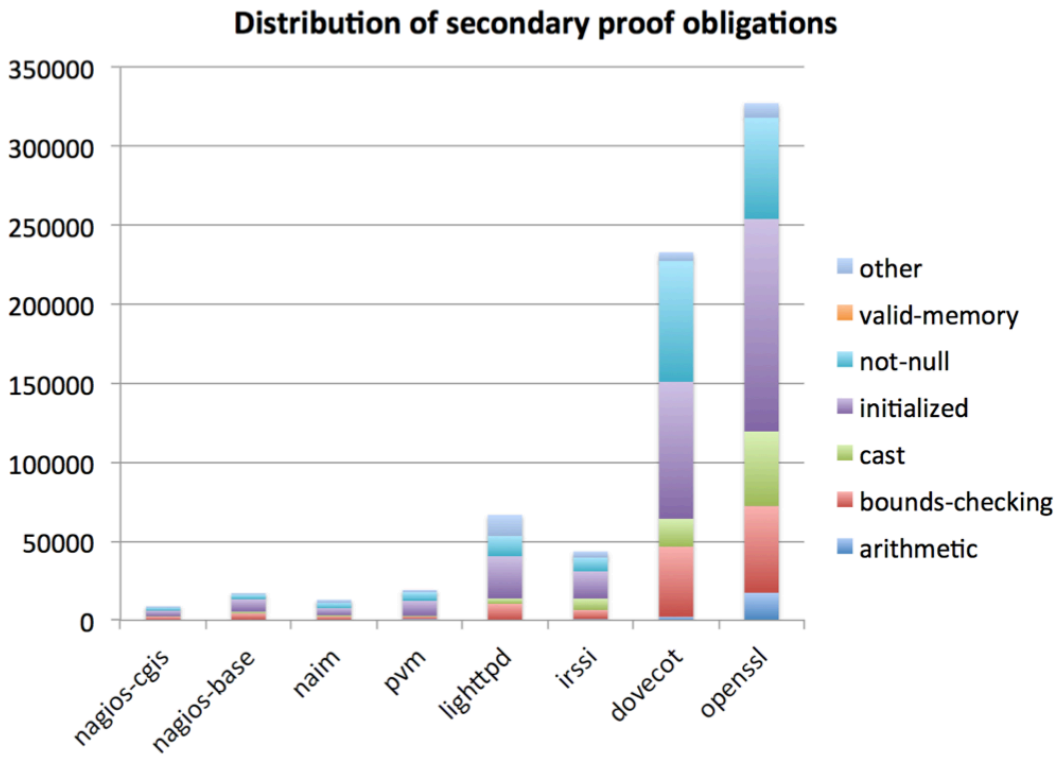
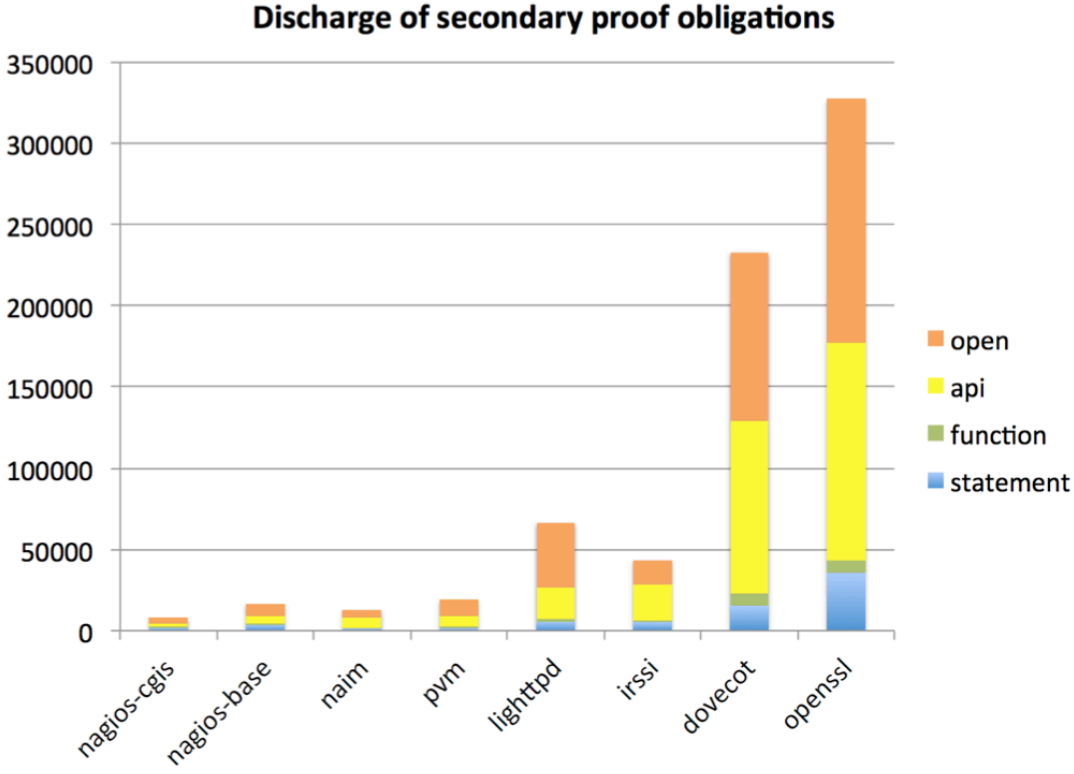


Figure 2 below summarizes where these primary obligations were eventually discharged for each test application. *Statement* refers to the obligations that were discharged by invariants at their original statement locations, *function* refers to the obligations that were discharged after being lifted to one or more containing functions, and *api* refers to obligations that were discharged after being lifted to a data structure api. *Open* refers to the obligations that could not be automatically discharged by CodeHawk. These represent either actual memory vulnerabilities, or locations that must be proved safe by hand.



Figures 3 and 4 below summarize the number, placement and discharge of secondary proof obligations for the 8 test applications.


Distribution of secondary proof obligations
350000
300000
250000
200000
150000
100000
50000
0
nagios-cgis
nagios-base
naim
pvm
lighttpd
irssi
dovecot
openssl
other
valid-memory
not-null
initialized
cast
bounds-checking
arithmetic


Discharge of secondary proof obligations
350000
300000
250000
200000
150000
100000
50000
0
nagios-cgis
nagios-base
naim
pvm
lighttpd
irssi
dovecot
openssl
open
api
function
statement

D. Record and track any discrepancies found using the problem reporting process.

Discussion   This element would only be relevant for an actual qualification exercise.