Tool Operational Verification and Validation Cases and Procedures 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 Cases and Procedures document for this analyzer as specified by DO-330 10.3.3. 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 cases and procedures that might be the content of an actual Cases and Procedures 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 10.3.3. It is based on a particular set of test cases and testing procedures that were previously used to fulfill a DHS research contract for the tool's development.

2 Cases and Procedures Framework

[From DO330-10.3.3] **The Tool Operational Verification and Validation Cases and Procedures detail how the tool operational verification and validation process activities are performed). This data should include:**

A. <u>Review and analysis procedures</u>: The scope and depth of the review or analysis methods to be used, in addition to the description in the Tool Verification Plan.

Formal methods tools are often characterized by essential properties that cannot be verified through testing. These are properties of soundness -- that an incorrect result will *never* be produced, or completeness -- that *all* instances of some condition will be identified. Requirements that a tool will do something, or won't do something, lend themselves to specific test cases that will demonstrate instances of the required positive or negative behavior. *All* or *never* requirements, however, can only be verified by proof because it is never feasible to test every possible combination of inputs to the tool. Since DO-330 tool qualification centers around verification-by-testing, the qualification of formal methods tools with soundness or completeness requirements must somehow be fit into this testing framework.

The requirements we list for the CodeHawk C Analyzer in the separate *Tool Requirements (TR) Framework for CodeHawk C Analyzer* document form a formal argument for the completeness of CodeHawk with respect to the possible program locations where memory safety may be compromised. The requirements jointly specify

the set of proof obligations, and their placement in the program, that must be discharged to support an inductive proof of memory safety over the sematic transfer function defined by the C language standard. A verification that this unsafe memory space is exhaustively covered by the proof obligations would require a human analysis of the CodeHawk source code with respect to the C standard. No amount of testing can verify this. Testing can establish whether any particular memory violation in the input C program is identified in the CodeHawk output by an undischarged proof obligation, or conversely, whether any particular C program location that CodeHawk deems safe is indeed safe in the input C program. Testing can also *refute* the claim of completeness (or soundness with respect to proved-safe locations) by finding a counter example. But finding no counter examples in any particular test run does not verify that none exist in all other runs.

The best one can do, then, is to expose CodeHawk to a sufficient *variety* of C program constructs to build confidence in the completeness requirement. It is beyond the (funding) scope of this research project to create, execute, and review an actual set of qualification test cases for CodeHawk. Rather, we are charged with identifying issues that may arise when subjecting formal methods tools, such as CodeHawk, to such qualification testing. Accordingly, we will discuss an actual set of test cases that were used for formal verification of CodeHawk in a related DHS research contract (Gold Standard), and contrast this with another set of test cases, the NSA Juliet Test Suite, that one might alternatively use in a tool qualification. Both attempt to assess the completeness of C memory safety analysis, but they take substantially different approaches to achieving sufficient input variety.

The Juliet Test Suite [1], first developed by NSA and now available from NIST, is a collection of over 81,000 small programs with embedded flaws meant to illustrate 181 Common Weakness Enumeration (CWE) instances. Each program exhibits one CWE and is usually paired with a similar program that does not contain the flaw. The suite comes with test drivers to automate the test execution and check, and count, the correct identifications (true positives), and misidentifications (false positives). The total suite functions as a verification of completeness (tool finds all 181 CWEs) and soundness (tool produces no false positives) with respect to the space covered by the 181 CWEs. This sort of granular, made-to-order isolation of individual flaws makes it possible to trace individual test cases to individual requirements, *if* the requirements were enumerated according to the CWE taxonomy. This would help achieve the DO-330 goal of traceability (DO-330: 6.1.5) but, as we have discussed in *Tool Operational Requirements (TOR) Framework for CodeHawk C Analyzer*, section E.1, the CWEs themselves do not form a rigorous or comprehensive enumeration of C memory safety. The CWEs were conceived as an informal enumeration of software bugs, and thus the Juliet Test Suite can be effective in assessing how many of these bugs can be found by a tool. But completeness (and soundness) with respect to the CWE taxonomy says very little about completeness and soundness relative to C memory safety.

It also appears that the fine-grained traceability that comes from small, toy programs competes with the confidence that a tool could find the same flaws in much larger, more complicated, real-world programs. One of the unique challenges of static analysis is

detecting the net program semantics embedded in complex control flow, function pointers, data aliasing and indirection. Analyzers that find simple instances in simple programs are often unable to find these same instances in many real program contexts. But building confidence in real program analysis requires real programs as test cases, and these in turn embody many overlapping and intertwined instances of memory compromise, defeating the traceability goal of one test case per one requirement. This is not so much a feature of formal methods tools as it is a feature of program analysis tools that take whole programs as input. An important aspect of whole program analysis is dealing with the arbitrarily complex, recursive variety of program forms. Achieving appropriately complex input variety, to warrant confidence in completeness and soundness, competes with tracing requirements to specific inputs and outputs. Human analysts must comb through the aggregate results of a tool and relate them to corresponding locations in the input program. This sort of testing is hard to automate, and very expensive in terms of human resources.

This was the motivation behind the DHS Gold Standard project that the CodeHawk C Analyzer was originally built for. To obtain an objective metric of the complete set of memory flaws in a set of real-world C applications, against which other analyzers could eventually be assessed, this human cost must be paid at least once. We began with a formal definition of the space of C memory safety and built an analyzer around the inductive enumeration of this space. Memory safety proofs automatically produced by the analyzer needed to be checked by humans against the actual safety of the input C constructs. Proof obligations that could not be automatically discharged (because the computed abstract invariants weren't sufficiently precise) were manually checked to determine whether they were in fact unsafe, or whether they could be proven safe with stronger invariants. This lead to humans adding the missing lemmas that would complete the proofs until all remaining unproven conjectures were determined to be false.

After this very expensive human verification exercise achieved a sound and complete enumeration of the space of memory flaws in 7 real-world C applications (the Gold Standard), future verification of other C analyzers could be automated against this enumeration in the style of the Juliet suite. The upshot here for D0-330 qualification of C analyzers is that *subsequent* tools could possibly reuse the Gold Standard test applications *plus* the canonical enumeration of their memory safety space as a way to gain confidence in soundness and completeness through automated testing. But the first qualification lacked both (full) automation and individual traceability.

As mentioned in the *Tool Requirements (TR) Framework for CodeHawk C* Analyzer document, the requirements we have listed there, and would thus address with test cases and procedures in this document, relate externally visible input/output results – requirements that specific classes of input constructs are diagnosed with output proofs (or non-proofs). Such requirements could be satisfied by analyzers using internal technologies other than abstract interpretation. In reference to DO-330 5.2.1.2(k), we noted other possible sets of requirements that might be defined to specify the correct implementation of internal CodeHawk technologies, including such things as abstract domains, program abstraction, fixpoint convergence, proof obligation placement and discharge. We will not further discuss test cases and procedures for these additional kinds

of requirements other than to note that such internal testing may indeed be feasible in the case of CodeHawk because of the external visibility of many of its intermediate results in the form of persistent xml files. Test cases and procedures could be built around the individual modules of CodeHawk with their immediate xml inputs and xml outputs. Had CodeHawk not been implemented with this sort of open architecture with visible intermediate results, it is not clear how one would go about separately testing the implementation of its internal technologies.

B. Test cases: The purpose of each test case, set of inputs, conditions, and expected results to achieve the pass/fail criteria.

Assuming that we were to use the Gold Standard test suite as the test cases for qualification, the test cases would consist of the following seven open source C applications.

Lighttpd

Lighttpd (www.lighttpd.net) is a web server designed and optimized for high performance environments. The application is medium sized, consisting of 89 source files, 846 function, and 52,058 lines of code. It is a well-designed program that nevertheless poses many challenges for analysis. The features that make it hard to analyze include its heavy use of complex, heap-allocated data structures with many implicit relationships between its members that must hold for the program to be safe.

Nagios

Nagios (www.nagios.org) is a monitoring system that enables organizations to identify and resolve IT infrastructure problems, designed with scalability and flexibility in mind. The application is medium sized, consisting of 65,133 lines of code. It builds into several separate executables. Although not as well designed as lighttpd (for example, it includes a large number of global variables) it is nevertheless easier to verify than lighttpd, because it has many fewer function inter-dependencies. The main challenge posed by the nagios applications is the size of some of the functions, some more than 4000 lines of code with complex control flow structure.

Naim

Naim (http://naim.n.ml.org) is a console client for AOL instant messenger (AIM), AOL I seek you (ICQ), Internet Relay Chat (IRC), and The lily CMC. Naim has a plugin architecture that supports the inclusion of third-party plugins. The application is medium sized, consisting of 29 source files, 715 functions, and 23,210 lines of code. The main challenge for analysis is its heavy use of function pointers to support the plugin architecture.

Pvm

Pvm (http://www.csm.ornl.gov/pvm/) (parallel virtual machine), developed by Oak Ridge National Labs, is an application that permits a heterogeneous collection of computers hooked up by a network to be used as a single large parallel computer.

Irssi

Irssi (www.irssi.org) is a terminal-based IRC client that uses a text-mode user interface.

Dovecot

Dovecot (www.dovecot.org) is an open-source IMAP and POP3 email server for Linux systems.

OpenSSL

OpenSSL is an open-source implementation of the Secure Sockets Layer and Transport Layer Security protocols combined with a full-strength general-purpose cryptography library. A large number of Internet-facing applications rely on openSSL for secure communication. The application made headlines in April 2014 when it was discovered to be vulnerable to an information leak due to a memory over-read: the Heartbleed vulnerability.

The purpose of each of these test cases is to expose the analyzer undergoing qualification to a representative sample of real C program constructs and memory vulnerabilities. Because the test cases are C applications, and CodeHawk is a C program analyzer, the inputs are the programs themselves. If this were the first use of these test applications, pre-Gold Standard, there would be no a priori expectations about pass/fail criteria. The applications were chosen to represent the variety of programming constructs and memory weaknesses that one might find in actual practice. The expectation is that CodeHawk should identify all locations that cannot be proven memory safe. In addition, it would be expected that all locations CodeHawk claims to be safe actually are safe. The number and nature of memory vulnerabilities in the input applications would not initially be known, and thus the initial CodeHawk testing would itself be part of the process that determines the actual vulnerabilities contained in the test applications.

On the other hand, if this were a qualification exercise performed post-Gold Standard, the seven applications would be accompanied by a machine-readable enumeration of the nature and location of each actual memory vulnerability. The expectation is that the analyzer undergoing qualification would correctly identify all, or some sufficiently large subset, of the actual vulnerabilities, and report no (or few) other locations as vulnerable. A qualifying analyzer with any false positives would be shown to be unsound, and one with any false negatives would be shown to be incomplete. Finding no false positives and no false negatives would not verify either soundness or completeness, but would raise confidence in the possession of these formal properties.

C. Test procedures: The step-by-step instructions for how each test case is to be set up and executed, selected inputs, how the test results are evaluated, and the test environment to be used.

The Gold Standard test applications would be set up and executed using their normal build Makefiles – the same procedures one would go through to build the applications with a compiler/linker. Pre-Gold Standard, the CodeHawk results would need to be examined by analysts and compared to the corresponding locations in the C input code to determine that "proved safe" locations are safe and that undischarged proof obligations represent either actual memory vulnerabilities or constructs whose safety conjectures cannot be proven at the precision level of the abstract domains used for the run.

Post Gold Standard, it would be possible to automate the results evaluation by having the computer check (against the Gold Standard benchmark) that all true vulnerabilities have been found and that no false ones have.

References

[1] Tim Boland and Paul E. Black, *Juliet 1.1 C/C++ and Java Test Suite*, IEEE Computer (2002), pp. 88-90.