Ballista Design and Methodology

October 1997

Philip Koopman
Institute for Complex Engineered Systems
Carnegie Mellon University
Hamershlag Hall D-202
Pittsburgh, PA 15213
koopman@cmu.edu
(412) 268-5225

Ballista Design and Methodology

October 1997

Philip Koopman
Institute for Complex Engineered Systems
Carnegie Mellon University
Hamershlag Hall D-202
Pittsburgh, PA 15213
koopman@cmu.edu
(412) 268-5225

Abstract

This report serves as initial documentation of the design and methodology to be employed by Ballista, an automatic robustness testing and hardening tool for Commercial Off-The-Shelf software components. The Ballista architecture includes the following major components:

- Automatic testing of software modules determines whether they behave robustly in the face of exceptional inputs.
- Characterization of response regions (behavior when presented exceptional input values) is performed over the parameter input space. Initial heuristics for this characterization are presented here.
- Software wrapper routines are created to deflect any exceptional inputs that would, if uncaught, produce a "crash" or "hang" of the application software component.
- A World Wide Web interface to users.

Introduction

Ballista is a project to automatically test commercial off-the-shelf (COTS) software for and harden against robustness "bugs" (*e.g.,* "hangs" and "crashes"). In particular, the emphasis is on proper detection and handling of exceptional conditions so that their occurrence does not compromise the integrity of system operation. Ballista is a three-year program started in October, 1996. This report serves as an explanation of the preliminary design of the Ballista architecture at a point one year into the project.

Ballista will harden software components by exercising software modules with combinations of valid and invalid inputs, then generating "wrapper" shells to filter dangerous inputs out before they reach the encased software module. Ballista works solely at the module interface level; neither access to nor modification of source code is required by our underlying approach. Additionally, functional specifications are not required. Figure 1 shows a conceptual picture of the operation of Ballista.

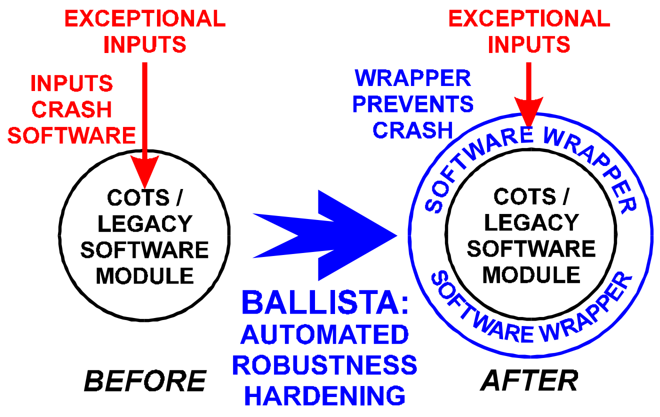

Figure 1. Ballista creates a software wrapper that filters out exceptional inputs from COTS software modules.

Ballista Architecture

The Ballista architecture will, when completed, be able to harden software components submitted to a World Wide Web site. In order to accomplish this, Ballista incorporates the following major components, depicted in detail in Figure 2.

- Automatic testing of software modules determines whether they behave robustly in the face of exceptional inputs.
- Characterization of response regions (behavior when presented exceptional input values) is performed over the parameter input space. Initial heuristics for this characterization are presented here.
- Software wrapper routines are created to deflect any exceptional inputs that would, if uncaught, produce a “crash” or “hang” of the application software component.
- A World Wide Web interface to users.

The following subsections detail the function of the various architectural components of Figure 2. The **module under test** in Figure 2 indicates that a single software module having a parameter-list interface is hardened by Ballista.

Robustness Testing

Robustness testing is automatically accomplished with software that generates test cases for the module under test Ballista uses an object-oriented approach centered on the data types contained in the module interface parameter list. The **data type definitions** are pre-defined data type information that provides base test cases, information about how to conduct random tests, and templates for use in assertion generation. Ballista comes pre-seeded with a variety of data type definitions in the form of an inheritance hierarchy (*e.g.,* “file pointer” inherits from data type “int” and “pointer to buffer area” inherits from “generic pointer”). Thus, information from the interface declaration is provided in terms of defined data types.

Base test cases consist of a set of manually-defined heuristics for testing a particular data type (the heuristics are accumulated over the inheritance hierarchy for that data type). For example, NULL is a value that heuristically is tested for all pointer values. These base test cases serve to find single-point discontinuities in the module response space in an efficient and effective manner.

A **random test capability** is available to provide a randomly-distributed coverage of the module response space. It generates test cases in a pseudo-random manner (and thus is repeatable by resetting the pseudo-random generator seed), and provides starting points for the pattern recognizer to begin building response regions.

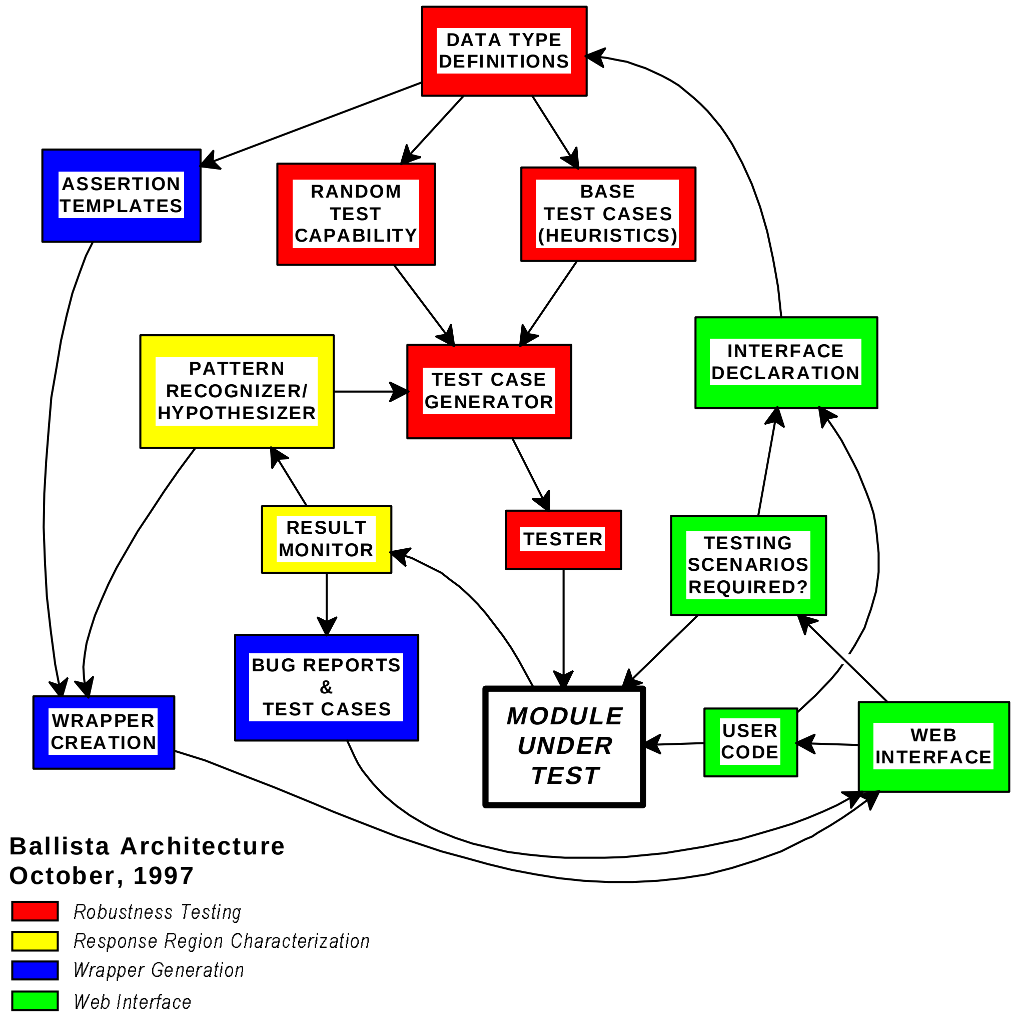

Figure 2. The Ballista architecture includes robustness testing, response region characterization, wrapper generation, and a World Wide Web interface.

The **test case generator** has two modes of operation. In open-loop mode it selects both base test cases and a number of random test cases to test the module and report any robustness bugs found. In closed-loop mode it interacts with the pattern recognizer to map response regions.

The **tester** is a test harness that executes the module under test with a particular test case. In particular, the tester initializes any required data structures, executes the module under test, and then erases any remaining data structures after the test.

Response Region Characterization

For each test case that the module under test executes, it either terminates gracefully or generates a Catastrophic, Restart, or Abort failure (defined in [Koopman97]). The **result monitor** tracks the status of the module under test and the computer system executing that module with a watchdog timer and task status queries looking for crashes or hangs. Results with other than graceful behavior can be reported as bugs, and are fed to the pattern recognizer.

The **pattern recognizer/hypothesizer** builds a map of the *response regions* (defined in [Koopman97]) of the module under test. It does so by using random and heuristic base test cases as a starting point to provided seeded response values. It then builds upon these values by "growing" regions of valid, Catastrophic, Restart, and Abort behaviors across the entire *n*-dimensional input parameter space. Because it is in general impractical to test all values of inputs for any particular module, the pattern recognizer uses statistical techniques to generate approximate response region maps. There is a tradeoff between execution time and confidence level in the accuracy of the response regions.

In general it is not possible to guarantee 100% accuracy. However, the use of heuristic base cases significantly reduces the likelihood of missing single-point discontinuities in the response space by exploiting knowledge about common programming bugs and how they interact with hardware protection mechanisms.

In order to keep the scope of this generation of Ballista technology tractable, the response region characterization only considers "static" robustness gaps (*i.e.,* it tests only inputs for a single function call, and does not attempt to find faults in finite state machines or persistent variable values than may exist within the module under test).

Wrapper Generation

Once a module has been tested and response regions have been characterized, a protective wrapper is synthesized to protect the module. The protection will be done with multi-dimensional assertions, which are executable statements that verify a parameter value is within acceptable ranges. Part of the data type definitions include **assertion templates**, which are templates that contain information about how to build assertions for a particular data type.

The **wrapper creation** function combines information about response regions with the assertion templates for each data type used by the module under test. It collapses tests into efficient range-check style assertions (as opposed to an exhaustive process that has a separate test for every possible input value, which would obviously be to large or too slow to be useful). The wrapper creation function produces C source code that compiled with or linked to the module under test to produce a hardened resultant module. In other words, if the wrapper is called instead of the original module under test, the result will be more robust in the face of exceptional input parameter values.

Web Interface

While the underlying Ballista technology does not require a web-based interface, the World Wide Web is being used as the deployment vehicle for the current project. A high-level **Web interface** permits the user to enter COTS component **user code** and a list of input parameter data types. If any supporting code is required to set up a **test scenario**, the user provides that as well. (Ballista need not have access to module-under-test source code; in fact it does not have such access in the operating system test application. However, this access greatly simplifies compilation and execution and so is used as an expedient for the current project).

Part of the information entered with the **user code** is information about the data types that form the interface to the module (**e.g.,** the first parameter might be a 32-bit integer while the second parameter might be a pointer). This information is used to drive testing and hardening via the data type definition information within Ballista.

Response Region Heuristics

The current test literature suggests that software bugs typically have relatively simple (*e.g.,* [Beizer95], pp. 159-160). However, we found at least one response region space that seemed to have complex structure as reported in [Koopman97]. In order to understand whether this poses a general problem, we have explored other response regions. Our conclusion is that there seem to be many functions with complex response regions, and that more than a simplistic single-variable approach is called for in a significant number of cases. For example, Figure 3 shows a response region generated for the function "fgets" on a Sparcstation.

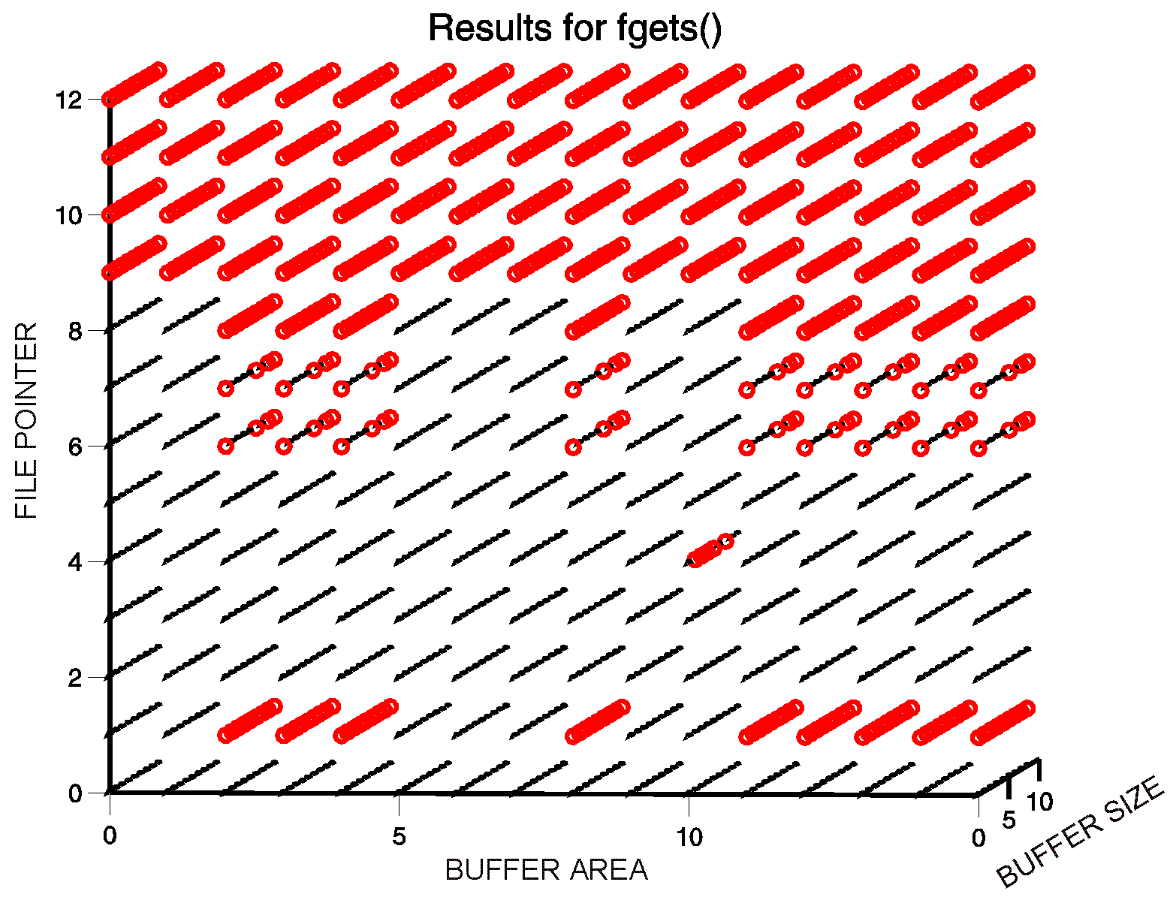

Figure 3. The fgets() function on a Sparcstation displays non-trivial structure in response regions. Circles represent Abort failures, while dots represent non-failure cases. The data are for various combinations of file pointer, buffer area, and buffer size parameters, with axis values indicating which test number is used, not the actual test value.

Given the potential complexity of response regions with respect to multi-dimensional data, we plan to evaluate the ability of heuristics in defining contiguous response regions. The importance of heuristics becomes even more important when large response areas are explored that are too large in size to be exhaustively tested. In cases where the search space is intractable, sampling and pattern recognition strategies will be applied as necessary.

Initial Response Region Heuristics

Below are the initial response region heuristics to be explored in the second year of the Ballista project. Combinations of these heuristics will probably be necessary to achieve both efficiency and accuracy.

- **Single-dimensional response.** In some cases a single value of data on a single parameter will produce a uniform response value. In Figure 3 this is represented by, for example, rows having all dots or all circles for a particular file pointer value.
- **1.5-dimensional response.** This case appears as horizontal or vertical "lines" of responses in Figure 3 that do not span the entire graph; for example the circles in a horizontal row for file pointer test

case 1. In this case one parameter is fixed to a set value, and a second parameter is varied over a limited range of values.

- **2-dimensional response.** This case appears as rectangular areas on the response region graph, and may potentially encompass multiple single- and 1.5- dimensional areas. In 2-dimensional response a subset of values for two input parameters encapsulate a response region.
- ***n*-dimensional response.** This is simply a generalization of 2-dimensional response. However, it may be computationally intractable and therefore not used in most cases.
- **Response region convex hull.** A set of randomly generated response tests of the same value may be encompassed using a convex hull algorithm. This can create a polygonal response region if that is deemed more accurate than a rectilinear response region.
- **Response region growing.** An initial "seed" value is obtained by randomly selecting a set of input parameters to test. Whatever response value is obtained is "grown" into a region by successively testing adjacent points in the response points in a pattern to encompass an homogenous response space. This heuristic may be used in conjunction with dimensional response determination by, for example, using a computer graphics algorithm to locate the periphery of a rectangle given a single point inside the rectangle. Note that both faulty and fault-free response regions can be grown in this manner.
- **Confidence assessment.** A response region may be created without necessarily having sampled all the interior points to be assured of homogeneity (in general, testing each point in the input space for response characteristics will be prohibitively expensive in terms of execution time). Therefore random or patterned points within a candidate response region may be tested to build confidence that the interior points are in fact homogeneous.
- **Discontinuity searches.** The fact that not every point in a response space can be tested means that, like any search strategy, response region determination will be vulnerable to missing single-point discontinuities in the response space. This will be overcome to a degree by using hand-selected base test cases to look for commonly occurring discontinuities. For example, NULL pointer values will be tested, as will MAXINT for integer values.

Future Plans

In the second year of the project we plan to develop automated error response region consolidation software in order to permit efficient screening of input parameters for potentially dangerous inputs (*i.e.*, inputs that will crash the module to be hardened). We will also select a demonstration application composed of many modules.

In the third year we plan to make the Ballista hardening service publicly available on this web site. The effectiveness of Ballista will be quantified by comparing the system-level robustness of the selected demonstration application both with and without module hardening.

References

[Beizer95] Beizer, B., Black-Box Testing, John Wiley & Sons, New York, 1995.

[Koopman97] Koopman, P., Sung, J., Dingman, C. & Siewiorek, D., "Comparing Operating Systems using Robustness Benchmarks," Symposium *on Reliable Distributed Systems*, October 1997.

Acknowledgement:

This research was sponsored by DARPA contract DABT63-96-C-0064 (the Ballista project).