RAVE Architecture: The Design of a System for Metrics-Based Security and Reliability Visualization

Paul Gestwicki and Joseph Morris
Computer Science Department
Ball State University

December 14, 2007

Abstract

We present the architecture and usage guide for RAVE, the Reliability Appraisal and Vulnerability Evaluation tool. This tool is useful for metrics-based analysis of software systems, and it has been tested on large-scale projects. RAVE uses a two-process architecture, separating the metrics collection utility from the graphical front-end. The user interface applies a novel approach to exploring a system, specifically by employing three views of the system: an Artifact View, Metrics View, and Visualization View. The Visualization View demonstrates the novel use of an interactive, zoomable radial call graph. The usage guide explains how RAVE can be configured and used, and the architectural notes provide direction on how to augment the existing system. Potential directions for future work are provided.

1 Introduction

Software security and reliability are related issues driving many modern research programs in software engineering. *Software reliability* refers to a software systems' capacity for maintaining correct operation in both routine and hostile circumstances. *Software security* refers to a software system's defenses against attacks that would cause the system to perform undesired operations. For example, a distributed denial of service attack tests software's reliabiity, and an SQL-injection attack tests software's security. Both security and reliability can be compromised through implementation and design errors [6]. Net-centric systems necessarily have a larger *attack surface* than non-networked systems [4].

domainis not within the scope of this work. The field of *software metrics* represents one approach for quantifying software quality. McCabe's metrics, one of the earliest and most well-known works in the field, are used model structured programs as graphs. The metrics relate properties of the graph to predications about software quality [5]. Zage metrics represent a different approach to metrics-based software analysis, and their design metrics have been shown to predict problem areas in large commercial systems [7]. Modern software

analysis tools include both the classical and more modernized and specialized versions of software metrics.

Recent research at the Software Engineering Research Center (SERC) has explored the relationships between software metrics, security, and reliability. This work has been conducted by the SMART Team and funded by Army Research Labs. One of the fundamental hypotheses of this work is that security and reliability are intrinsically related, specificially that a security vulnerability is a reliability flaw. The team's research so far supports this hypothesis, although it became clear through this work that a tool was required to support continued analysis.

This technical report describes RAVE, the Reliability Appraisal and Vulnerability Evaluation tool, which is designed to address the needs of the SMART team. It illustrates a three-view approach to exploring software systems. The *Metrics View* displays the quantitative metrics analysis of a software system. The *Artifact View* provides more specific details about a specific module by showing the electronic artifact corresponding to the module. In the most common case, the module is a function and its artifact is its source code, although more generally an artifact can be a script, configuration file, or database. The *Visualization View* provides a means for a user to navigate through a system using modern techniques of zoomable user interfaces.

2 User's Guide

This section provides an overview of how to launch, configure, and use the RAVE system. RAVE requires access to the source code for a system and, ideally, the libraries on which it depends. RAVE also requires a UDC file describing the software system, and this file can be generated using one of SciTools' *Understand* tools. We have tested RAVE using specifically *C++ Understand*, although similar tools exist for other programming languages. This software is available at `http://www.scitools.com`, and it comes with detailed instructions for generating a UDC file. Note that the UDC file must be generated using absolute paths, not relative paths. This also means that the workstation for running RAVE must be the same workstation on which the system's source code is stored and from which the UDC file was generated.

2.1 Launching and Configuring RAVE

RAVE is packaged as an executable jar, and so it can be launched by either double-clicking its icon (on most operating systems) or from the command line:

```
java -jar rave.jar
```

RAVE uses the standard Java logging facilities provided in the `java.util.logging` package. This facility are accessed through the SLF4J framework, which is available from `http://www.slf4j.org` under an MIT License. Logging is disabled by default, but because the standard logging facilities are used, the logging settings can be configured by specifying the "java.util.logging.config.file" or "java.util.logging.config.file" system property, as described in the JDK documentation for the `java.util.logging.LogManager` class.

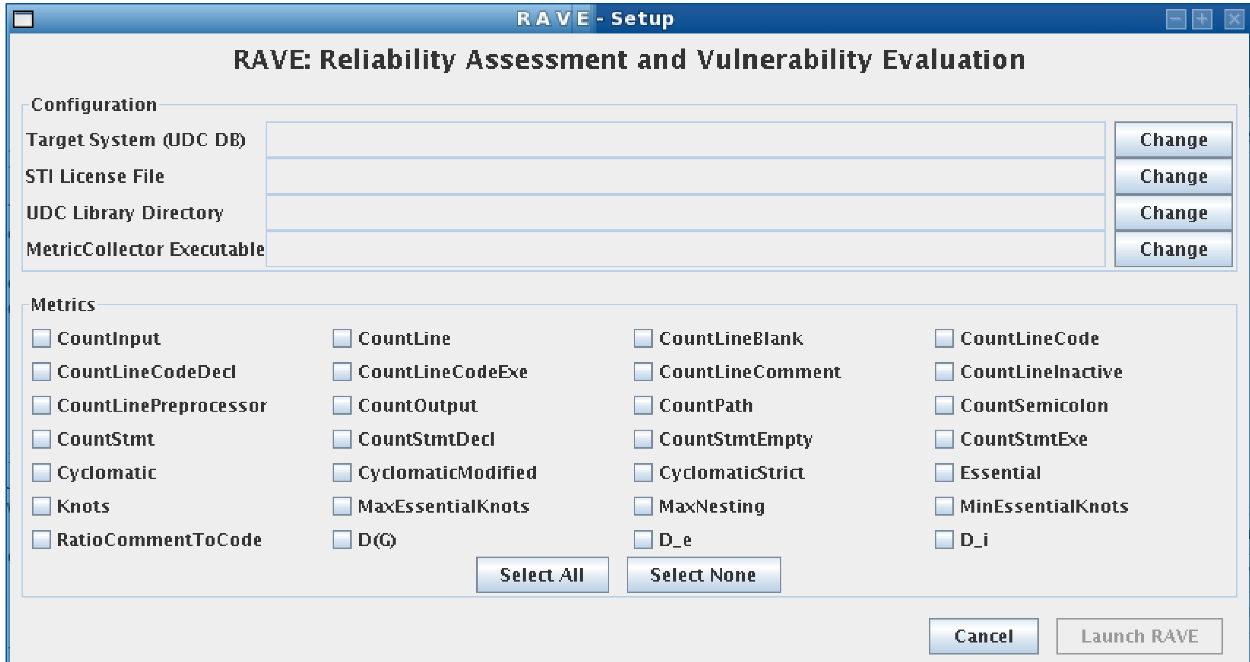


Figure 1: RAVE Configuration Dialog

After a brief initialization, a configuration dialog box will open. A screenshot of this dialog box is shown in Figure 1. The upper portion of the configuration dialog box is used to specify the system under investigation as well as RAVE's dependencies. RAVE stores this configuration using Java's preferences system, so the selections will already be in place the next time RAVE is launched. Table 1 describes the values to use in this portion of the configuration dialog.

The bottom portion of the configuration dialog allows the user to choose the set of metrics to compute on the selected system. At least one metric must be chosen before the user may continue. The three metrics in the lower-right — D(G), D_e, and D_i — are the Zage metrics design complexity, external complexity, and internal complexity [7]. The other metrics are the primitives provided by the Understand tool and include McCabe's cyclomatic complexity [5]. The definitions of these are provided in the tools' manuals, which are publicly available online. It should be noted that once the configuration is closed and RAVE proper is launched, metrics cannot be recomputed; however, when running RAVE again, more metrics can be chosen.

2.2 The Three Views

RAVE displays a system using three views: artifact view, metrics view, and visualization view. Each view is embedded into its own window, and the intention is that RAVE can be run on a three-monitor workstation, with one window dedicated to each view. Of course, RAVE can be run with less monitors using standard window management tools to navigate the three views.

By default, selections are "linked" among the three views. That is, selecting a module

Target System (UDC DB):	This is Understand database file for the system under test.
STI License File	This specifies the location of the user's STI license file, which is required to use the Understand runtime libraries.
UDC Library Directory:	This is the location of the runtime libraries required by RAVE. The setting here is platform-dependent. On Linux, this should be the `udb_util` directory in the UnderstandCollector distribution. On Microsoft Windows, this is the directory that contains the release executable for UnderstandCollector.
MetricCollector Executable:	This is the platform-specific executable to run the UnderstandCollector tool.

Table 1: Required configuration for RAVE

in one view will cause the corresponding module to be selected in the other two views. This synchronization empowers a user to explore a complex system using whichever view is appropriate for the specific task. For example, a user may wish to start by selecting the module with the highest D(G) value from the metrics view: this selection is mirrored in the visualization and the artifact view. The user then sees an unexpected neighbor in the visualization call graph and so selects this neighbor. Immediately, the source code is shown in the artifact view and its metrics analysis is shown in the metrics view. At any time, the user may unlink the selections using the button at the top of the visualization view window.

The *Artifact View* shows the file in the filesystem that is associated with the selected module. As shown in Figure 2, the Artifact View uses two panels to show the file's contents. The left panel is a standard tree view of a filesystem, and expanding a source file depicts all of the modules defined within it. The right panel displays the contents of the file, such as the source code shown in the figure. Modules' source code is not always available, for example, for library methods referenced in header files. In such cases, a message is shown in the right panel that there is no artifact definition for the module.

The *Metrics View* shows the values for all of the metrics computed for a system. Figure 3 is a screenshot of the Metrics View. The leftmost column lists the names of the modules in a system, and each subsequent column contains an evaluated metric for the corresponding module. The default listing of metrics is alphabetical, but columns can be dragged into different positions by the user. Clicking on a column heading will sort the entire table by that column in either ascending (first click) or descending order (second click).

The Metrics View is used to specify which modules should be marked as potnential vulnerabilities. A context menu can be acquired at module names in the table; in most operating systems, the action associated with context menus is a right-click. Figure 4 illustrates the context menu as has been used to mark the module `BSDgetopt` as a potential vulnerability. Vulnerable modules are shown in red throughout the table and in the Visualization View.

Visualization View depicts the structure of a software system as a radial call graph. A screenshot is shown in Figure 5 Only one level of depth is shown in the call tree by default. Increased levels up to three are user-selectable through the spinner at the bottom of the Visualization View. Figure 6 shows a visualization with two levels of depth shown. The

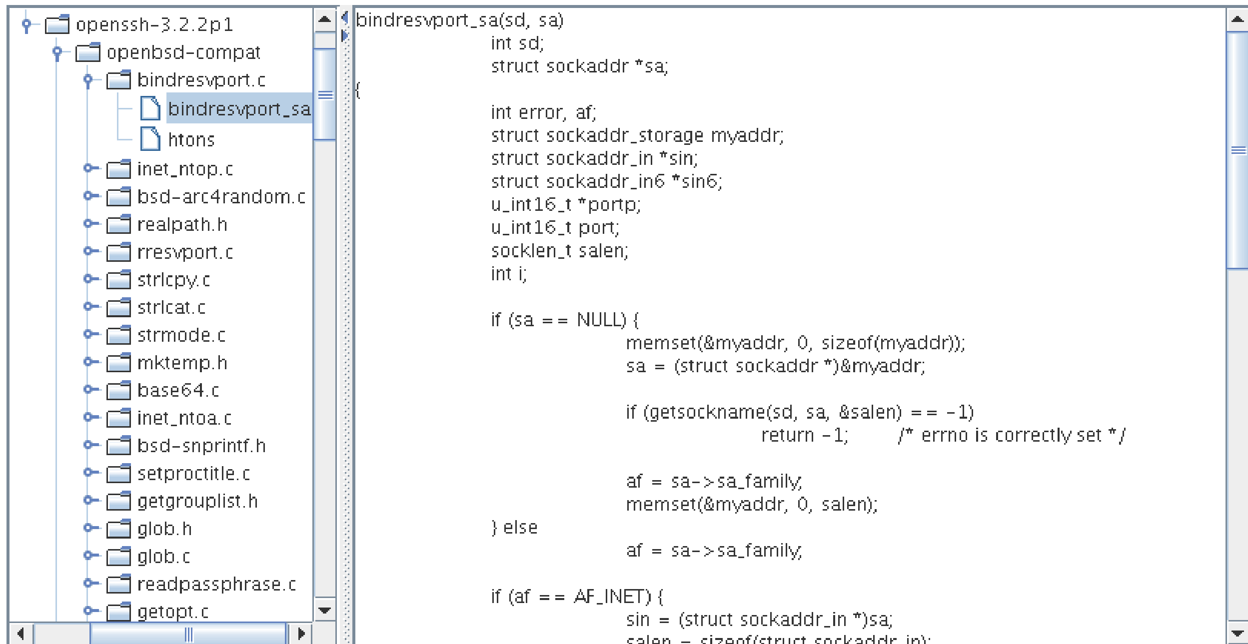


Figure 2: Artifact View

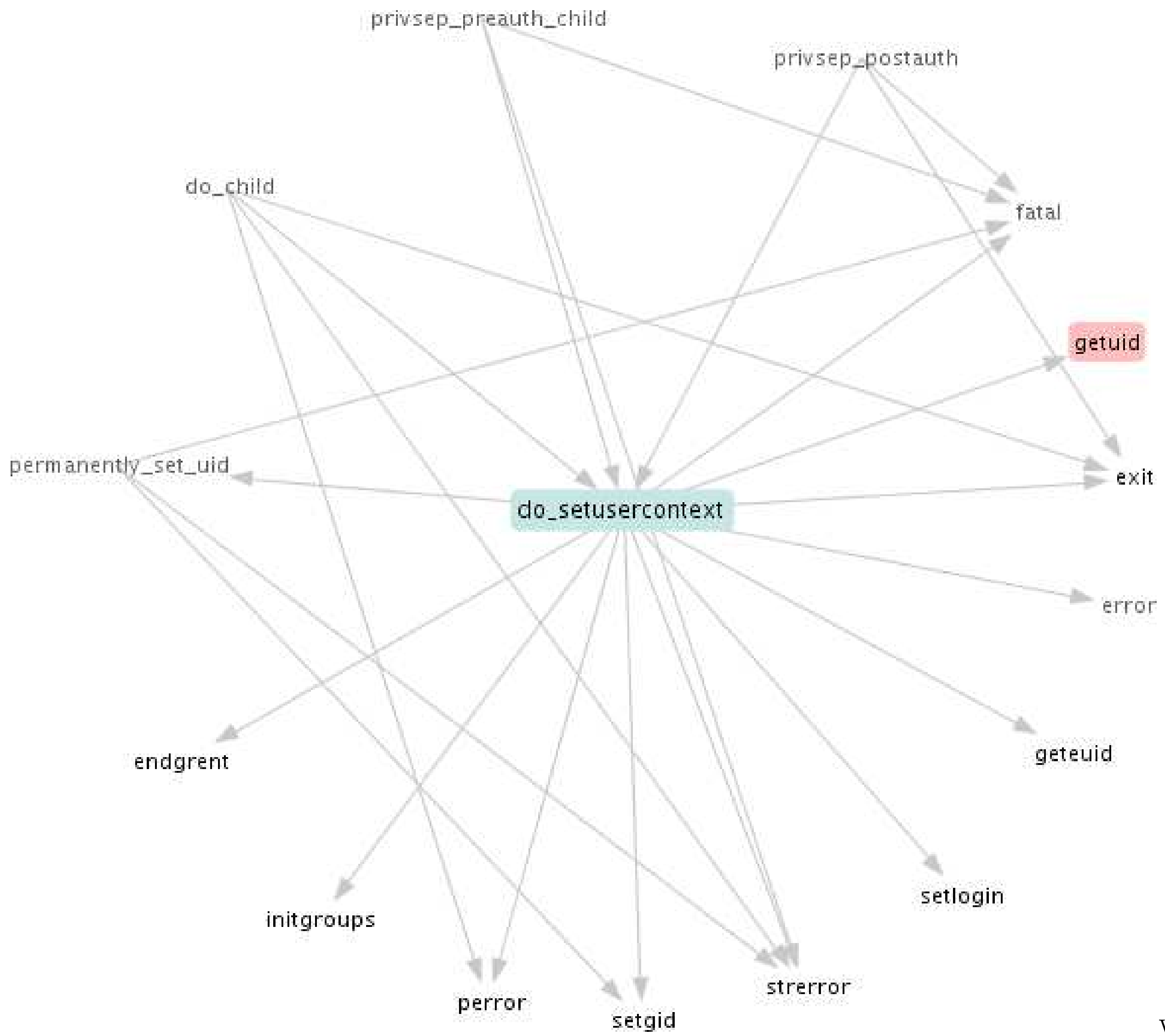
Module	CountInput	CountLine	CountLineB...	CountLine...	CountLine...	CountLine...	CountLine...	CountLinel...
buffer_get_string	62.0	20.0	0.0	15.0	4.0	9.0	5.0	0.0
buffer_get_short	1.0	7.0	0.0	7.0	3.0	2.0	0.0	0.0
buffer_get_int64	4.0	7.0	0.0	7.0	3.0	2.0	0.0	0.0
buffer_get_int	63.0	7.0	0.0	7.0	3.0	2.0	0.0	0.0
buffer_get_char	33.0	7.0	0.0	7.0	3.0	2.0	0.0	0.0
buffer_get_bignum_bits	3.0	12.0	1.0	11.0	4.0	7.0	0.0	0.0
buffer_get_bignum2	10.0	9.0	0.0	8.0	4.0	3.0	1.0	0.0
buffer_get_bignum	10.0	17.0	1.0	14.0	4.0	8.0	2.0	0.0
buffer_get	16.0	9.0	0.0	9.0	2.0	5.0	0.0	0.0
buffer_free	95.0	6.0	0.0	6.0	2.0	2.0	0.0	0.0
buffer_dump	6.0	15.0	1.0	14.0	4.0	8.0	0.0	0.0
buffer_consume_end	6.0	7.0	0.0	7.0	2.0	3.0	0.0	0.0
buffer_consume	32.0	7.0	0.0	7.0	2.0	3.0	0.0	0.0
buffer_compress_uninit	9.0	16.0	0.0	16.0	2.0	12.0	0.0	0.0
buffer_compress_init_s...	5.0	11.0	0.0	11.0	2.0	7.0	0.0	0.0
buffer_compress_init_r...	4.0	8.0	0.0	8.0	2.0	4.0	0.0	0.0
buffer_compress	6.0	35.0	4.0	24.0	4.0	17.0	7.0	0.0

Figure 3: Metrics View

Module	D(G)	CountInput	CountLine	CountLineB...	CountLi...
main	364.0	4.0	80.0	12.0	57.0
mm_free	556.0	16.0	74.0	11.0	57.0
choose_dh	273.0	8.0	64.0	7.0	57.0
do_log	81.0	17.0	59.0	2.0	57.0
BSDgetopt	100.0	18.0	62.0	1.0	57.0
mm_send_keyst[illegible]	[illegible]	[illegible]0.0	78.0	17.0	56.0
mm_answer_key[illegible]	[illegible]	[illegible].0	72.0	14.0	56.0
process_authentication...	1211.0	8.0	67.0	7.0	55.0

Vulnerable
Non-vulnerable

Figure 4: Vulnerability selection in Metrics View



width
6.1in

Figure 5: Visualization View. Shown here is the call tree for `do_setusercontext` from OpenSSH 3.2.2p1. `getuid` has been marked as potentially vulnerable and so is highlighted in red.

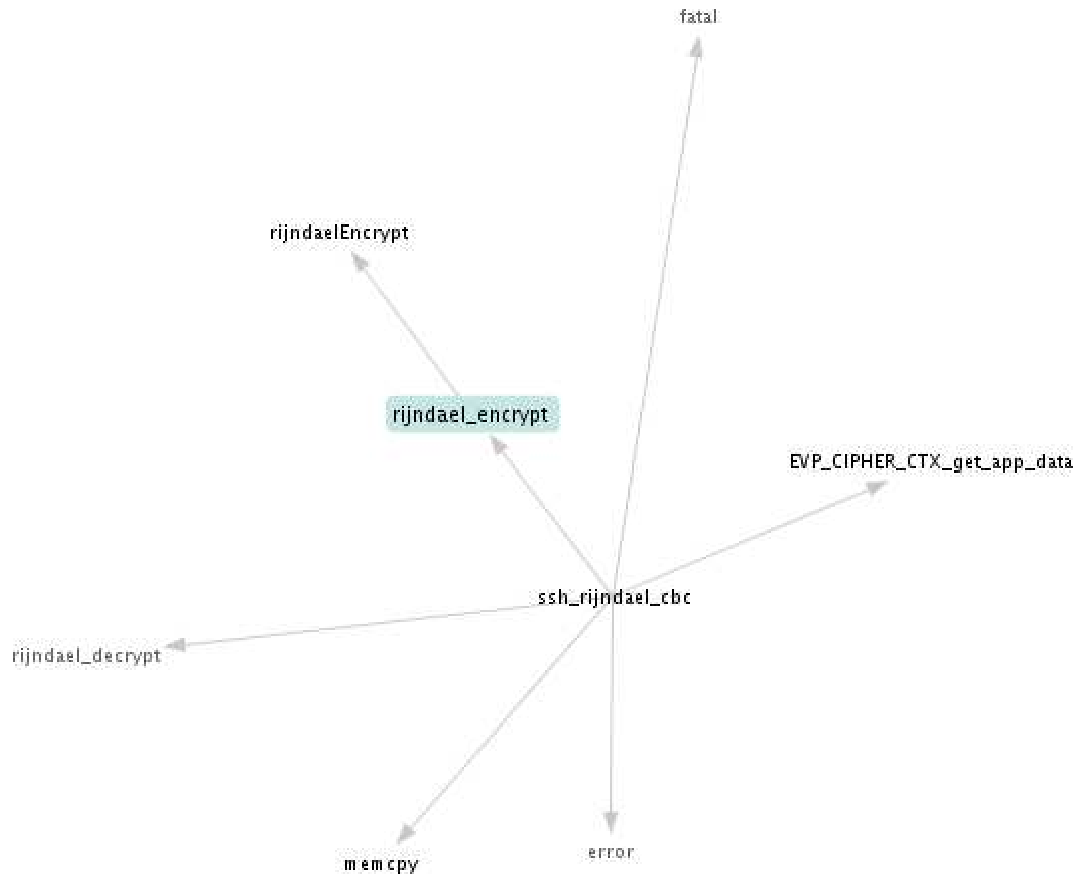


Figure 6: Visualization View showing two levels of depth in the call tree. This call tree is rooted at `rijndael_encrypt` from OpenSSH 3.2.2p1.

figure was chosen for clarity's sake; in practice, for nontrivial software systems, depths of two or three may become too complex to be usable, and our experimentation showed that four or more levels was not worth supporting at all.

The mouse is used to navigate through the visualization. Pointing to any node (without clicking) will cause its name to be shown in the lower-left of the view. This is useful when navigating complex structures or attempting to select one node from a cluster. Left-clicking on any node will select it and move it to the center of the visualization. As the figures have shown, the selected node is shaded. Left-clicking outside of any node allows the user to pan the visualization. Right-clicking and dragging upwards or downwards will zoom the visualization out or in, respectively. A right-click without moving the mouse will fit the visualization optimally into the window's bounds, which is useful when resizing windows or zooming.

2.3 Best Practices

The Metrics View provides the best entrypoint into understanding a system. A user may select as many metrics as desired during the RAVE configuration, but the ones that have anecdotally proven the most useful in our analysis are D(G) and the Cyclomatic metrics, which have been shown in previous work to correctly identify stress points in software. It is recommended that a user start by exploring the Metrics View, sorting the modules by metrics of interest. The top few modules for each metric then can be manually identified as potential vulnerabilities. Of special interest are those modules that rank highly in multiple metrics. After identifying these potential vulnerabilities, the Visualization View can be use to explore the relationships among these modules. Specifically, one may look for neighborhoods that have several potential vulnerabilities identified. Also, modules adjacent to several potentially vulnerable modules merit further inspection. Finally, the Artifact View can be used to get a fine-grained details about a module. A user may identify a vulnerability in the artifact itself, but RAVE is not designed to be an editor or integrated development environment; a user should use the appropriate IDE for the software project to document the vulnerability and make the appropriate fixes. It is obviously not feasible to create a single vulnerability visualization tool that can seamlessly integrate into all development environments and processes.

Although RAVE has not undergone rigorous user testing, the developers and SMART team researchers have established current best practices for using RAVE for systems analysis. RAVE is designed to be executed on a workstation with three display devices, and we have found that this is necessary in order to get the most benefit from the Visualization View specifically. The other views can easily be resized to fit, for example, side-by-side on one display. In fact, the Metrics View is rather unwieldy to have maximized in the same amount of space as makes the Visualization View usable. Hence, two monitors should provide a comparable experience to our three-monitor testing. When restricted to a single display device, a user may consider maximizing the Visualization View while tiling the other two, although this may cause some loss in efficacy in the Visualization View animations of selection changes.

3 Metrics Collector

RAVE's metrics collector is as a stand-alone tool with an interactive command-line interface. The metric collector takes commands from the standard input stream and outputs responses directly to the standard output stream. The metric collector then is connected to the RAVE frontend via the Pipes and Filters architectural design pattern [1]. The metrics collector interfaces directly with the Understand libraries and the UDC database at runtime. It was written in C++ using the Standard Template Library and the Understand API.

The metrics collector as a stand-alone tool be used in an interactive style or run in a batch mode. The metrics collector will run in batch mode if it is given the following command-line arguments: a UDC file, a mode string to select the types of entities for which to collect metrics (currently "functions" or "structures", although "structures" is untested), and a file containing the names of functions known to vulnerable in the system. This list of files is necessary for certain metrics not supported by the RAVE front-end, such as neighborhood metrics. In the absence of these command-line arguments, the metrics collector enters interactive mode by default. For example, the following command will start the metrics collector in batch mode on Windows:

```
UnderstandCollector.exe project.udc functions project-vulns.txt
```

In batch-mode the system collects all supported metrics and write them to standard out in a tab-delimited format. The first line of the output contains the name of the entity, the file in which it is defined, and the names of all the metrics.

3.1 Communication Protocol

In interactive mode, any command entered by a user receives a response. All commands are case sensitive. The response will be either the command's output, an "OK" message when the command produces no output, or an error message. All error messages begin with "ERROR:" and are followed by the error description. The output of a successful command execution will vary depending on the command. Every command must print some value so that the RAVE front-end knows when to execute the next command. Entering a command involves entering the command name followed by any required arguments and a newline. For example, the following command opens the named UDC database:

```
OPENDB project.udc
```

3.2 Commands

3.3 General Usage

Table 2 lists all the commands and their corresponding arguments. The METRIC command has a special syntax since it requires a named metric and a list of arguments for the metric. The format of the METRIC command follows:

```
METRIC metric_name[arg1, arg2, ..., argN]
```

Command	Arguments	Description
METRIC	Metric call string	Parses and executes a given metric string.
HELP	Metric name	Prints information for the usage of a given metric.
OPENDB	Name of a UDC file	Opens the UDC file and initializes the database.
CLOSEDB	None	Closes any open databases.
EXIT	None	Exits the program.

Table 2: Commands recognized by the RAVE metrics collector.

Metric Name	Description
und_c	STI Understand built-in metrics.
calltree	Prints out the call tree.
neighborhood	Computes neighborhood metrics.
zage	Computes various Zage metrics from STI Understand primitives.

Table 3: Descriptions of supported metrics.

If the metric requires no arguments, then the square brackets are optional. Supported metrics are shown in Table 3, with their arguments described in Table 4.

Most of these metrics return a scalar result. However, `calltree` does not. It outputs the call tree of the system. The format for the call tree is as follows:

```
module_name     understand_identifier     file:start_line-end_line
     | called_module1     understand_identifier
     | called_module2     identifier
     ...
```

As an example, consider this output from an OpenSSH tree dump:

```
grace_alarm_handler     c#AF5B6D0F4CAF7A93 openssh-4.5p1/sshd.c:349-357
     | kill     ckill
     | sigdie     csigdie@file=openssh-4.5p1/log.c
     | get_remote_ipaddr     cget_remote_ipaddr@file=openssh-4.5p1/canohost.c
```

The call tree output is terminated by an empty line.

3.4 Architecture of the Understand Collector

The metrics collector architecture is divided into three major components: a command processing layer, a metric abstraction layer, and a proxy for the Understand library. The metrics collector was designed for extensibility, allowing for the addition of new and non-scalar metrics as well as the replacement of the STI Understand subsystem with another, such as the Design Metrics Analyzer. To support this, lower tiers of the architecture typically return their results as abstract `util::StreamPrinter` objects to upper tiers. The `StreamPrinter` interface provides only one method, a `print` method that print its contents to a C++ `ostream`. To prevent memory leaks, objects are often returned to upper tiers encapsulted

Metric Name	**Arguments**
`und_c`	**module**: typically a function name **metric**: name of a metric supported by Understand
`calltree`	No arguments required
`neighborhood`	**type**: currently, only "function" is supported **mode**: choose "vuln" to count vulnerable modules in neighborhoods, or choose "total" to count the total number of modules **file name**: a text file containing the names of vulnerable modules, one per line **depth**: number of levels to recurse in the call tree
`zage`	**module**: typically a function name **metric**: one of "cc", "de", "di", "dg"

Table 4: Descriptions of supported metrics.

Namespace	**Description**
`command_util`	User command processing.
`metric_util`	Metrics abstractions and the metrics registry.
`metrics`	Implementations of metrics.
`udb_util`	Proxy to access the Understand library.
`util`	Common utilities used throughout the system.

Table 5: Namespace responsibilities in the RAVE metrics collector.

by smart pointers such as `std::auto_ptr` and, if supported by the STL implementation, `std::tr1::shared_ptr` objects.

The various namespaces in the metrics collector are shown in Figure 7, and an overview of these namespaces is provided in Table 5. Command processing is done in the `command_util` namespace. When a command is entered from standard in, the input is read and passed as a stream into the `CommandParser` class. `CommandParser` is a factory which parses the stream and returns the appropriate `Command` object. All commands in the system inherit from the `Command` interface, which has an `execute` method returning a `StreamPrinter` object. Executing a command and printing its result is done with the following sequence of commands:

```
auto_ptr<Command> cmd(CommandParser::parse(cin)); //retrieve command object
auto_ptr<StreamPrinter> pr(cmd->execute()); //get result of command execution
cout<<*pr<<endl; //print the result
```

The metrics abstraction layer can be found in the `metric_util` namespace, with the actual metrics implementations separated into the `metrics` namespace. The `metric_util` namespace contains the `Metric` interface, from which all metrics must inherit. Each `Metric` takes a list of argument strings as input, computes the desired metric using the `udb_util::Udb` proxy, and returns a result. To prevent coupling to particular return value types and to allow for more flexible metrics, `Metric` objects only return abstract `Result` objects, subclasses of `StreamPrinter`. It is this property that has allowed for operations such as retrieving the

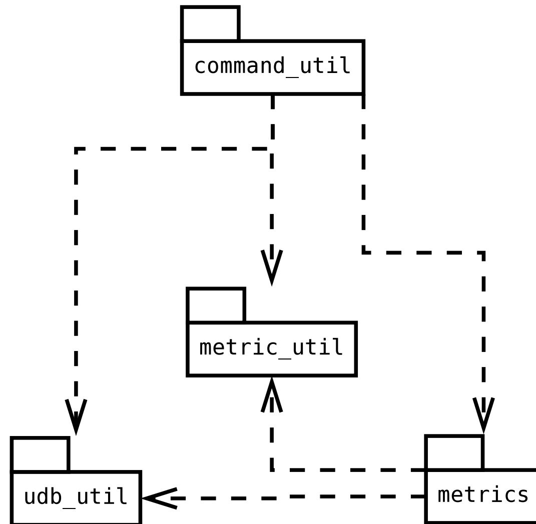


Figure 7: Dependencies among major namespaces in the metrics collector.

calltree to be implemented as `Metric` objects. All `Metrics` in the system are stored in the `metric_util::Registry` singleton class and are retrieved by name. This registry is reinitialized whenever a new database is opened, since some metrics cache information from the database upon construction to improve performance. Initializing the registry and adding metrics to it must be done by another class, specifically `command_util::OpenDB`.

The `udb_util` namespace contains the proxy class `Udb`. It is a necessary abstraction to work around linking issues with the Understand library under newer versions of the GNU toolchain. The Linux version of the Understand API library was statically linked against an older version of the GNU `libstdc++` libraries. Since we built the metrics collector with a newer version of the GNU toolchain, the C++ runtime (such as exceptions, etc.) did not work properly when the application was linked directly to the Understand library. This was because symbols were resolved to the wrong versions of the libraries at runtime. Under Linux, `dlopen` is used to load the library during runtime and enforce proper symbol resolution. Once it is loaded, all the Understand API functions are assigned to function pointers in this class. Under Linux `udb_util` must be built as a shared library to work properly, and this library must be placed in the working directory of the metrics collector during execution. Under Windows, this class acts as a simple wrapper around the Understand API functions.

4 RAVE Front-end

The RAVE front-end is a Java application written in JDK 1.6. It has been tested on Linux and Microsoft Windows XP. As of this writing, an early-access version of the Java Runtime Environment is required in order for RAVE to run on Windows; specifically, version 6u10 contains a bugfix required for RAVE to function properly. RAVE uses the prefuse library for information visualization and the SLF4J logging library. Both of these libraries are incorporated into the executable `rave.jar` file using the Ant build script (`build.xml`) provided in the RAVE source archive.

Package	Description
edu.bsu.smart.rave	Common top-level classes.
edu.bsu.smart.rave.artifact	Support for module artifacts such as program source code.
edu.bsu.smart.rave.graph	Integration with prefuse.
edu.bsu.smart.rave.io	Graph readers and filesystem support.
edu.bsu.smart.rave.metrics	Generic metrics support.
edu.bsu.smart.rave.metrics.mc	Metrics support specific to the UnderstandCollector.
edu.bsu.smart.rave.module	Generic support for modules within a system.
edu.bsu.smart.rave.ui	Custom user-interface widgets.
edu.bsu.smart.rave.util	General utilities used in other packages.
edu.bsu.smart.rave.view	Visualization views.

Table 6: Packages comprising the RAVE Front-End Implementation

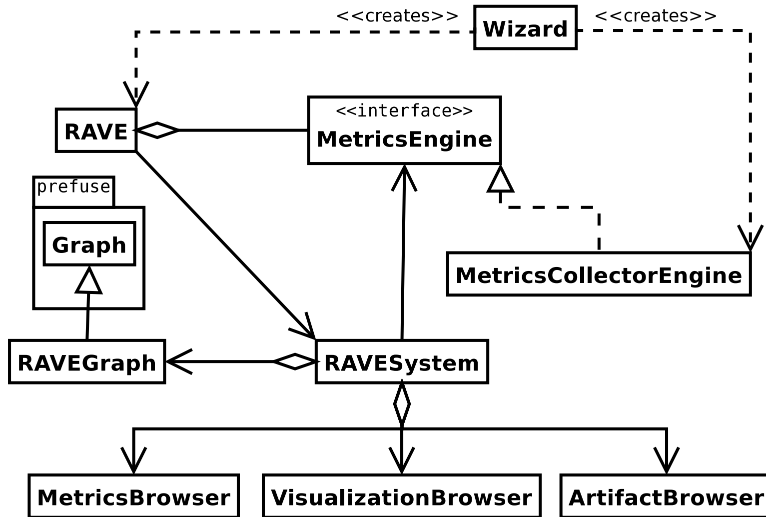


Figure 8: RAVE Front-end System Architecture Overview

The front-end launches the metrics collector as a separate process, specifically in the `MetricCollectorEngine`'s static `createProxy` method, and the two processes communicate through a pipe. That is, RAVE sends commands through the standard input stream of the metric collector and reads responses through its standard output stream.

The RAVE implementation is organized into ten Java packages. The separation of tasks is shown in Table 6; the packages' Javadoc documentation can be referenced for more details.

Figure 8 provides an overview of the front-end software architecture. The `Wizard` is the RAVE configuration dialog described in Section 2.1. It is responsible for creating the `MetricsCollectorEngine` as well as `RAVE` itself. Once RAVE is launched, the `RAVE` class creates its `RAVESystem` object, which represents the system under test. `RAVESystem` manages the three *browsers*, which to the user are the three views of the system described in Section 2.2. The primary data structure for the system itself is the `RAVEGraph`, a subclass of the prefuse `Graph` class.

Column	Description
NODE_NAME	The label for module represented by this node.
NODE_UID	A system-unique identifier for the node.
NODE_ARTIFACT_LOCATION	An absolute path on the filesystem to the node's artifact.
VULNERABLE	Indicates if this module is a potential vulnerability or not.

Table 7: Definition of the RAVEGraph Node table

Column	Description
EDGE_SOURCE	Row number of this edge's source node.
EDGE_TARGET	Row number of this edge's target node.

Table 8: Definition of the RAVEGraph Edges table

It is not the intent for this document to provide the equivalent of a code review for the entire RAVE front-end. The implementation has been carefully documented using the Javadoc standards, making it trivial to generate usable API documentation. However, certain aspects of the implementation may not be immediately clear to future developers and maintainers, due to unfamiliarity with referenced libraries or Java APIs, design patterns, or idiolectic artifacts. Two of these aspects are the RAVEGraph and the Visual Proxy Architecture, described in Sections 4.1 and 4.2 below.

4.1 The RAVEGraph Data Model

A *RAVEGraph* is the primary data structure for representing a system under analysis modulo the metrics analysis. It is implemented in the `RAVEGraph` class, which is a subclass of the prefuse `Graph` class. As such, a RAVEGraph is a specialization of a prefuse graph, and hence a RAVEGraph can take advantage of prefuse's four-tier data model [2]. However, the prefuse `Graph` class adopts a white-box design, which engenders increased coupling, decreased cohesion, and generally more expensive software evolution. Hence, within RAVE, access to the `RAVEGraph` class is limited as much as possible to its own package, and so other parts of the system (such as the Artifact View and Metrics View) never interact directly with the RAVEGraph data model.

A system under analysis is represented as a RAVEGraph. This graph's nodes and edges are represented as a pair of tables in which a tuple represents a single node or edge, respecitvely. This is independent of any visualization of the graph; that is, this only concerns the data of the system, not its view. Each column has a name that is represented as a Java `String`, and these constants are defined in the `RAVEGraph` implementation; we will use here the symbolic constants defined within the implementation rather their `String` values. The meanings of the columns for the nodes and edges tables are provided in Tables 7 and 8. All of the values in the table are immutable except for the node table's `VULNERABILITY` column, which is boolean and defaults to false but can be overridden through the Metrics View, as

Figure 9: Reification of the Visual Proxy Design Pattern in RAVE

described in Section 2.2.

Note that the row numbers referenced in the edge table refer to the node table, which implies that each node has two system-unique identifiers: its row in the table and its UID. The table row number is only used within the prefuse-integration and visualization subsystems, specifically the `io`, `graph`, and `view` packages. Outside of these domains, for integration with the other views of RAVE, modules are referenced by their UIDs.

4.2 Visual Proxy

The *Visual Proxy* design pattern is an approach to building user interfaces for object-oriented systems [3]. It is a specialization of the Presentation-Abstraction-Control (PAC) software architecture [1]. In the Visual Proxy pattern, each abstraction can create its own visual proxy (*i.e.* Presentation in PAC), and these proxies are composed into the user interface. This is notably different than the more commonly-known Model-View-Controller (MVC) architecture, in which the Model and View are separate parts of the system, and the Model must be attached to its View through a Controller [1].

The Visual Proxy reification in RAVE is based on the sample implementation in the original presentation of the pattern [3], as shown in Figure 9. The `UserInterface` interface provides one method, `visualProxy`, that returns a `JComponent` proxy for the object that implements `UserInterface`. The first parameter to `visualProxy` is the name of an attribute of the implementor. Throughout RAVE, in the classes that implement `UserInterface`, these attributes are specified by public constants. These follow the naming convention that they all end in "`_PROXY`," and they are the only such public constants. For example, `EvaluatedMetrics` implements `UserInterface` and defines one visual proxy attribute, `TABLE_PROXY`. The second parameter indicates whether the requested proxy should be read-only or allow mutability of the corresponding model. Hence, given an `EvaluatedMetrics` object `m`, a read-only graphical representation of it can be acquired by the following method invocation:

```
m.visualProxy(EvaluatedMetrics.TABLE_PROXY, true);
```

5 Conclusions and Future Work

This research has shown that three views of a software system can be combined into a usable system for a metrics-based exploration of software vulnerabilities. RAVE has led to some unexpected observations on our two test systems, Apache httpd and OpenSSH. However, more research is required in order to develop and evaluate a more formal process for integrating RAVE into systems analysis.

The primary visualization research contribution of this work is the use of a zoomable user interface for call graphs, specifically the multi-level radial call graph. The use of color annotations for potential vulnerabilities reveals relationships within the code in a fundamentally different way than classical top-down call trees. Formal usability testing is required to to quantitately relate our work to existing standards for call graph visualization.

Acknowledgements

This project was funded by Army Research Labs through the Software Engineering Research Center. Dr. Wayne Zage and Dolores Zage, both of Ball State University, were the principle investigators on this project. The RAVE splash screen was designed by Blake Self. The first three-view software prototype was developed by Dr. Ugo Buy and his research team at the University of Illinois at Chicago.

References

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, 1996.

[2] Jeffrey Heer, Stuart K. Card, and James A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430, New York, NY, USA, 2005. ACM. ISBN 1-58113-998-5. doi: http://doi.acm.org/10.1145/1054972.1055031.

[3] Allen Holub. Building user interfaces for object-oriented systems. Six part series in Javaworld, 1999–2000.

[4] Pratyusa Manadhata, Jeannette Wing, Mark Flynn, and Miles McQueen. Measuring the attack surfaces of two FTP daemons. In *QoP '06: Proceedings of the 2nd ACM workshop on Quality of protection*, pages 3–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-553-3. doi: http://doi.acm.org/10.1145/1179494.1179497.

[5] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[6] Joseph Morris. Identifying Potential Vulnerabilities in Software Design. Master's thesis, Ball State University, Muncie, Indiana, USA, December 2007.

[7] Wayne M. Zage and Dolores M. Zage. Evaluating design metrics on large-scale software. *IEEE Softw.*, 10(4):75–81, 1993.