Black-Box Fuzzing of the REDHAWK Software Communications Architecture

Shereef Sayed

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Jeffrey H. Reed, Chair
Carl Dietrich
Cameron D. Patterson

March 27, 2015
Blacksburg, Virginia

Keywords: Security, Fuzzing, SCA, OSSIE, REDHAWK

Black-Box Fuzzing of the REDHAWK Software Communications Architecture

Shereef Sayed

(ABSTRACT)

As the complexity of software increases, so does the complexity of software testing. This challenge is especially true for modern military communications as radio functionality becomes more digital than analog. The Software Communications Architecture was introduced to manage the increased complexity of software radios. But the challenge of testing software radios still remains. A common methodology of software testing is the unit test. However, unit testing of software assumes that the software under test can be decomposed into its fundamental units of work. The intention of such decomposition is to simplify the problem of identifying the set of test cases needed to demonstrate correct behavior. In practice, large software efforts can rarely be decomposed in simple and obvious ways. In this paper, we introduce the fuzzing methodology of software testing as it applies to software radios. Fuzzing is a methodology that acts only on the inputs of a system and iteratively generates new test cases in order to identify points of failure in the system under test. The REDHAWK implementation of the Software Communications Architecture is employed as the system under test by a fuzzing framework called Peach. Fuzz testing of REDHAWK identified a software bug within the Core Framework, along with a systemic flaw that leaves the system in an invalid state and open to malicious use. It is recommended that a form of Fault Detection be integrated into REDHAWK for collocated processes at a minimum, and distributed processes at best, in order to provide a more fault tolerant system.

Acknowledgments

Words are not enough to describe the profound impact that MPRG has left upon me. The work presented in this document represents the culimination of so many bright individuals that contributed to my growth as an academic and a professional.

I would like to take a moment to honor the memory of Jenny Frank. If the students are the lifeblood of the lab, and the faculty are the brains, then Jenny was the heart and soul. Her contributions to the success of the lab, the faculty, staff, students, and myself, are immeasurable. For all of her guidance and patience that she afforded me, I can only give thanks for her presence. Thank you so very much for everything that you have given me, Jenny.

My time at MPRG afforded me many great opportunities. The most memorable being the OSSIE project. What began as a singular effort went on to become a major open-source project that lasted ten years, with users all across the globe. While my contributions were significant, the project would not have succeeded without the contributions from so many others. I would like to thank Kathyayani Srikanteswara for her support and guidance during the earliest stages of what would later become OSSIE; Carlos Aguaya for his unwavering committment to a project that had every reason to fail; Rekha Menon, Karthik Channak, Chris Vander Valk, and Tom Tsou for all of the long hours and hundreds of lines of code.

Of course, OSSIE could not have existed without the ambitions of Dr. Max Robert. Under his leadership, a group of students worked for free to lay the foundation for a project that would last for ten more years. The qualities necessary to inspire people to accomplish what they did not know was possible can only be found in Max. None of this would have been possible without him. Max is my both my colleague and mentor, but always my closest friend.

In the end, all of this comes back to Dr. Jeff Reed. It was Dr. Reed that provided me the opportunity to work with Kathyayani, and it was Dr. Reed that encouraged Max and his students to pursue OSSIE. And it was Dr. Reed that afforded me the second chance I needed to finish what I started. Thank you, Dr. Reed, for all that you have given me.

These are just a few of the special individuals found only at MPRG. There are so many other talented faculty and students that contributed to the success of OSSIE. I would like

to thank Carl Dietrich for leading OSSIE and so many students to success; Philip Balister for creating a true open-source community for OSSIE.

Most of all, I give thanks to my wife, Sahana, for always believing in me and inspiring me to reach for new heights. I could not have completed this work and so much more without her by my side. It is no small wonder that my wife was also a student at MPRG.

Contents

List of Figures

Listings

Chapter 1

Introduction

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult." C.A.R. Hoare

The industry of writing software is unlike any technical discipline in that there are no defined metrics of right and wrong. The software developer is literally defining the laws of right and wrong in code. It is, therefore, inevitable that the resulting product will be biased towards failure because a single individual is responsible for distinguishing right from wrong. Just as the court of law interprets right and wrong by a jury of unbiased peers, software must be similarly tested for failure.

In our modern world of smartphones, the "Internet of Things", and "always on" connectivity, the need for better testing has never been greater. In particular, the need to identify and resolve security vulnerabilities has never been greater. From the perspective of the developer, it is not feasible to imagine the many ways that a system can fail or be exploited. Instead, it is preferable to let the machine identify failures and report them to the developer for evaluation to distinguish incorrect behavior from security vulnerability, of which there exists a very thin line of separation.

Testing software for failure is a difficult task because it deviates entirely from the classical algorithmic approach of writing software in the first place. There is no "quicksort" that exists for testing software. Instead, there are methodologies of testing software. A common method of software testing is to print the current state or location of the program. However, this method is primitive and risks introducing unwanted artifacts into the program. A more rigorous method of software testing is the unit test, in which each module is independently tested for correct behavior. However, unit testing assumes that the entire program can be deconstructed into independent units of work, when it is more likely that each unit depends on another. Each of these testing methodologies represent a form of positive software testing, in that they are intended to test for correctness. An alternative test methodology is a form

of negative software testing, which tests for failure.

An emerging method of negative software testing is called Fuzzing [17], which is based on the notion of adding random noise to a known pattern. Software fuzzing is a brute force testing strategy in which the inputs are continuously randomly generated until a failure is detected. In this paper, we will discuss different types of software fuzzing, the current tools available for software fuzzing, and demonstrate the value of software fuzzing by using REDHAWK as a case study.

Chapter 2

The Software Communications Architecture

The Software Communications Architecture (SCA) [6] has its roots in the SPEAKeasy [10] program, which was an effort sponsored by the United State Air Force (USAF) between 1990 and 1995 to develop and demonstrate a Software Defined Radio (SDR). By SDR, we mean a radio architecture that consists of an RF front-end, an ADC/DAC, and a microprocessor, such that all base-band processing is performed by the microprocessor, as shown in Figure 2.1. The goal of an SDR platform is to reduce the complexity of the radio design while maximizing reuse of the platform for virtually any type of RF communications. The SDR radio architecture presents several challenges that need to be overcome in order to realize the idealized goal of a true SDR. First, the RF front-end must be "tunable" across a broad range of the spectrum. The SPEAKeasy program was meant to operate between 2 MHz and 2 GHz. To achieve this, a variable Intermediate Frequency (IF) bandpass filter with low tolerance of error is required. Research in MEMS capacitors is leading the way into realizing a variable bandpass filter [3]. Second, the ADC/DAC must not only be "tunable", but must also be able to accommodate very high sampling rates in order to meet the minimum Nyquist sampling rate while maintaining a low signal-to-noise ratio (SNR). To achieve this, several "wideband" RF chip sets have been created that attempt to balance dynamic range with "good enough" SNR [5]. Research in compressed sensing techniques may hold the promise of a better ADC/DAC because it would allow the system to sample at sub-Nyquist rates [18]. Third, the microprocessor must be capable of supporting any required base-band digital signal processing (DSP) routines, such as Finite Impulse Response (FIR) filters, Fast Fourier Transforms (FFT), voice codecs, user interface, and system control. It is here, in the microprocessor, that the SCA resides.

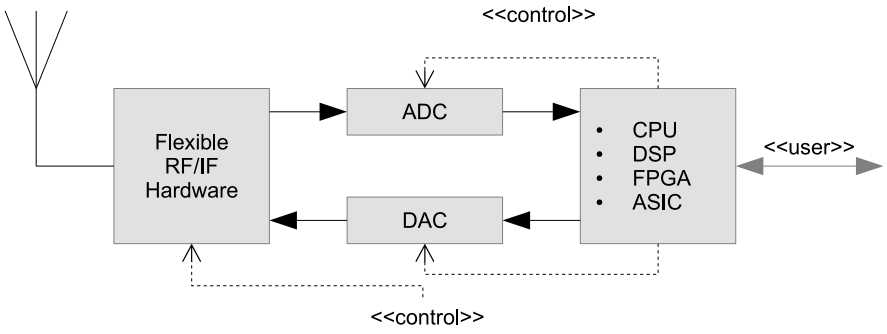

Figure 2.1: Block Diagram of an SDR Platform

2.1 Purpose of the SCA

The SCA is an open-standard specification commissioned by the Department of Defense (DoD) as the next iteration of the SPEAKeasy program. The Joint Tactical Radio System (JTRS) is a DoD effort to unify military communications, with the SCA as the foundation. While the JTRS program has suffered setbacks, the SCA specification continues to evolve as a standard [6]. The SCA was designed to leverage best practices in software development, which would allow it to find applications beyond the JTRS program.

The Core Framework (CF) of the SCA, shown in Figure 2.2, does not include functionality that is specific to a radio. In fact, the function of the CF is purposefully generalized so that the SCA may be extended in ways that are meaningful to any particular application space. Instead, the CF provides the developer with functionality that is common, and necessary, for any application space. Functionality such as a file system, file I/O, logging, configuration files, status information, and deployment requirements. To the communications engineer, the features of the SCA may appear to be unneccessary because they do not specifically address the challenges of designing and implementing a radio. However, this perspective misses the purpose of the SCA entirely.

The DoD procures a wide variety of technologies from a large number of vendors, such as fighter jets, tanks, transport vehicles, and firearms. However, no device is more prolific than the radio. Every fighter jet, tank, transport vehicle, even some firearms, and of course soldiers are equipped with some form of radio communications. Moreover, soldiers tend to be equipped with multiple radios, each with a different purpose. As radio designs have become more digital than analog [11], the DoD is procuring more and more software from disparate sources.

The purpose of the SCA is not to solve a radio design problem, but to solve a software management problem. In fact, the “Communications” in the SCA is not a reference to RF communication, but rather to Inter-Process Communication (IPC).

2.2 Design of the SCA

The SCA is a component-based design that is composed of two parts, the CF and the XML Domain Profiles. The CF is a software architecture which is composed of a heirarchy of system management and service components. The XML Domain Profiles are a set of configuration files that are provided as input to the CF components in order to configure the system to a known state. CF components communicate their state to each other through the Common Object Request Broker Architecture (CORBA) [14], which is a middleware technology that provides an abstraction of IPC mechanisms. The combination of the CF, XML Domain Profiles, and CORBA allow the SCA developer to create custom applications which maximize component reuse for new applications and different platforms.

2.2.1 CORBA

CORBA is the foundation of the CF, and all CF objects are derived from a CORBA interface module, which is defined using the Interface Definition Language (IDL). Each object in the CF posesses a set of interfaces (i.e. methods/functions) that are accessible as Remote Procedure Calls (RPC). The CORBA specification defines the General Inter-ORB Protocol (GIOP) to carry and route method invocations on CORBA objects. GIOP exists in the network transport layer, meaning that GIOP acts as the payload in existing protocols, such as TCP or UDP. Moreover, implementations of the CORBA specification may be extended to support new transport layers, such as shared memory.

CORBA is an open-standard specification of a middleware technology that is produced and maintained by the Object Management Group (OMG). Software developers are free to create implementations of the CORBA specification and distribute their implementation as they choose. Any meaningful implementation of the CORBA specification will include support for GIOP v1.0, v1.1, and v1.2, along with TCP and UDP transport mechanisms, and the CORBA NameService.

The CORBA NameService is a network service that provides object registration and lookup capabilities for other CORBA objects to access. An object registers with the NameService by "binding" its unique object reference to a readable name. Other CORBA objects may request access to an object reference from the NameService by asking for the object by its readable name. The NameService allows the CF to share and access different object capabilities while hiding their location from the developer.

2.2.2 Core Framework

The CF is a software architecture which is composed of a heirarchy of system management and service components. A diagram of the relationships between CF components is shown

in Figure 2.2. Only the relevant components of the CF will be discussed. Further discussion of each component can be found in the SCA specification [6].

The components of the CF are broken into four categories: *Base Application*, *Framework Control*, *Base Device*, and *Framework Services*.

The *Base Application* components are the following:

- CF::PropertySet
- CF::LifeCycle
- CF::TestableObject
- CF::PortSupplier
- CF::Port
- CF::Resource
- CF::ResourceFactory

Components of the *Base Application* category are intended to be implemented by the application developer [6]. The primary *Base Application* component is the CF::Resource, and is the basic unit of work for the application developer.

The *Framework Control* components are the following:

- CF::DomainManager
- CF::DeviceManager
- CF::ApplicationFactory
- CF::Application

The *Framework Control* components make up the heart of the CF. The CF::DomainManager is responsible for the registration and unregistration of CF::Applications, CF::Devices, and CF::DeviceManagers that exist within the domain [6]. The CF::DeviceManager is responsible for the creation of logical devices and launching service applications on these logical devices [6]. The CF::ApplicationFactory is responsible for the creation of a specific type of CF::Application in the domain [6]. The CF::Application that is created by the CF::ApplicationFactory maintains a list of CF::Resource components that make up a particular application.

The *Base Device* components are the following:

- CF::Device
- CF::LoadableDevice
- CF::ExecutableDevice
- CF::AggregateDevice

The *Base Device* components are logical representations of specific hardware device that exists in the platform. These logical representations are designed to be extended to meet the needs of the application on the target platform. However, the CF::ExecutableDevice is intended to be available in any domain because it is assumed that every domain will contain at least one executable device, namely the CPU that is hosting the SCA. The CF::ExecutableDevice is responsible for executing a given binary, which is typically a software process that extends the CF::Resource component.

The *Framework Service* components are the following:

- CF::File
- CF::FileSystem
- CF::FileManager

The *Framework Service* components provide basic file I/O and file management services within the CF. Moreover, because CF components are CORBA components, the *Framework Service* components are really providing a distributed file system within the domain. This is an incredibly powerful feature of the SCA.

For the purposes of this document, we will be focusing on the following CF components:

- CF::DomainManager
- CF::DeviceManager
- CF::ExecutableDevice
- CF::ApplicationFactory

These components represent the primary functionality of the CF within any implementation of the SCA.

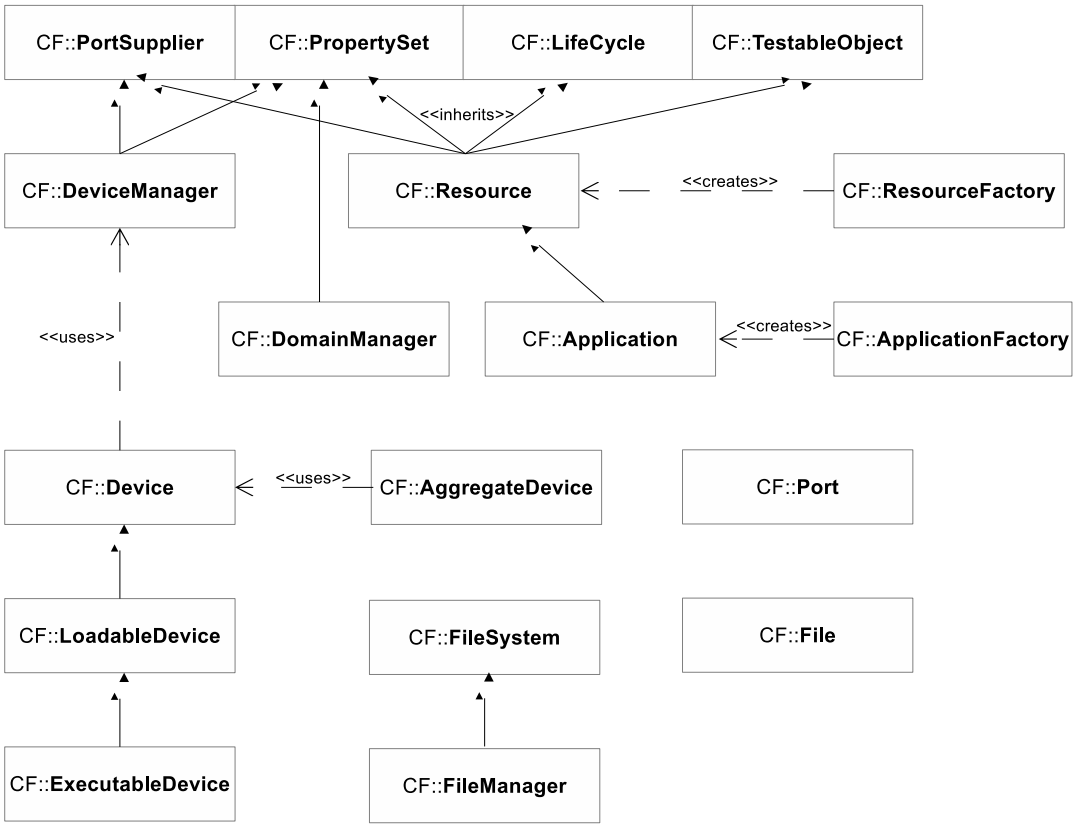

Figure 2.2: Core Framework IDL Relationships

2.2.3 Domain Profiles

The initial state of CF components is captured in a set of file descriptors referred to as the Domain Profile, shown in Figure 2.3. The properties, dependencies, and locations of each component that makes up an SCA system domain is defined within the Domain Profile as XML. The Domain Profile is the primary input to configure the domain to a known state, making them the focus of this document. The purpose of each file descriptor will be discussed, and further details can be found in the SCA specification [6].

The XML Domain Profile is composed of the following types of file descriptors:

- Domain Manager Configuration Descriptor (DMD)
- Device Configuration Descriptor (DCD)
- Software Assembly Descriptor (SAD)
- Software Package Descriptor (SPD)
- Software Component Descriptor (SCD)
- Device Package Descriptor (DPD)
- Properties Descriptor (PRF)

DMD: A DMD describes the set of services available to the domain, such as logging, along with the location of these services as found in the referenced SPD.

DCD: A DCD contains information about the devices associated with a device manager, how to find the domain manager, and the configuration information for a device [6].

SAD: A SAD contains information about the components that make up an application, and the application factory uses this information when creating an application [6].

SPD: A SPD contains information about a software components implementation. A software component can have multiple implementations for different target platforms. For example, a software component may be compiled for both ARM and x86 platforms.

SCD: A SCD describes the set of interfaces that make up a software component. Given the excessive use of multiple inheritance within the CF, the SCD provides a window into the basic functionality of a software component without the need to inspect the source code. However, the SCD also describes information about component ports, which is an advanced feature of the SCA.

DPD: A DPD describes basic information of a device such as manufacturer, model number, device family, etc. A set of DPD files will effectively describe all devices on a target platform.

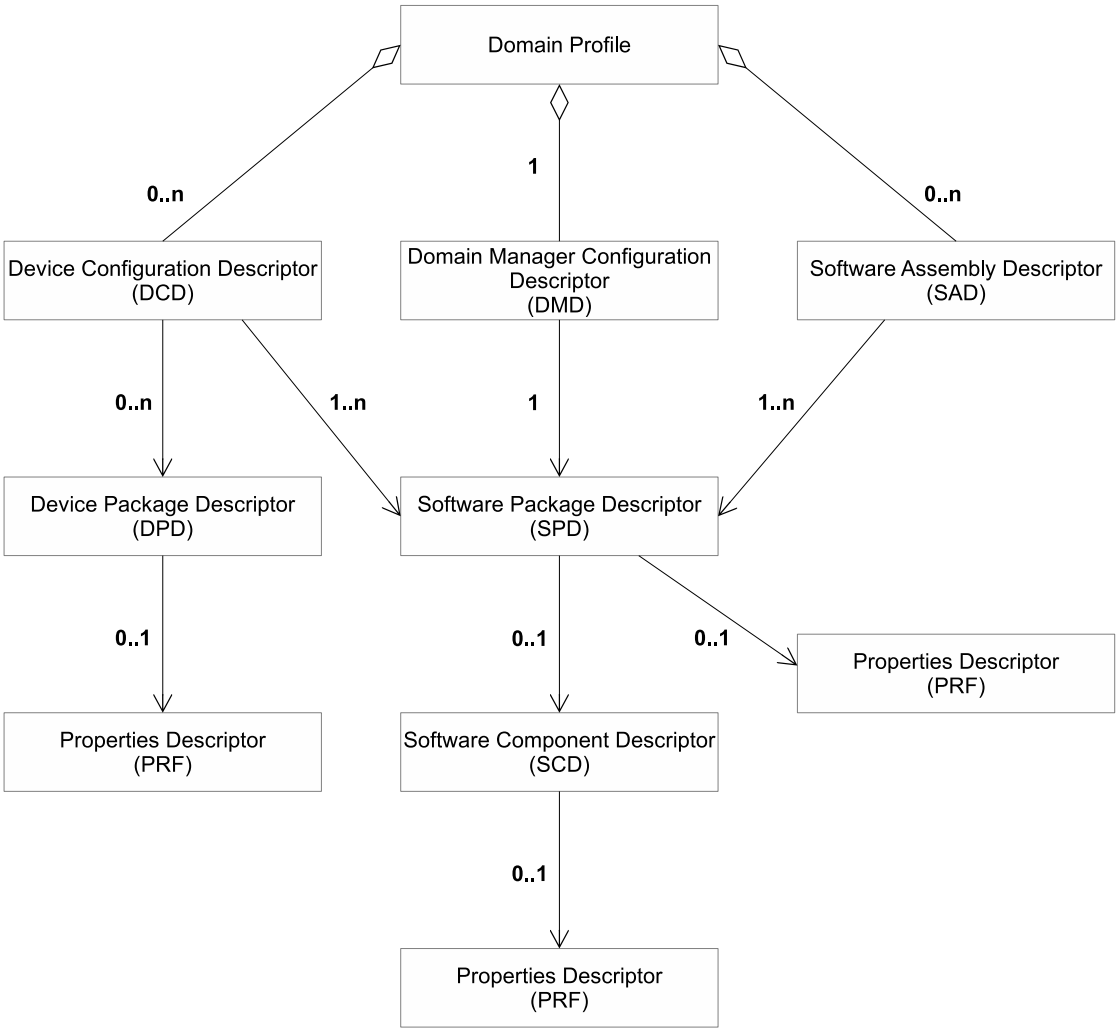

Figure 2.3: Relational Diagram of XML Domain Profiles

PRF: A PRF describes the properties of a software component as type-name-value trebles. For example, the PRF of an FIR filter component might contain the set of coefficients for each tap.

2.3 Implementation of the SCA

The SCA is an open standard that allows anyone to create an implementation of the specification. During the initial release of the SCA standard, the earliest implementations were either proprietary or commercial. Seeing a need for a community implementation, the SDR Forum (now the Wireless Innovation Forum; WInnForum) contracted Canada's Communications Research Centre (CRC) to create an SCA reference implementation, also known as SCARI. The CRC wrote SCARI in Java and tested it for compliance with the standard. However, the reference implementation was not well suited for application development because it lacked any form of developer support. This state of affairs within the SCA community motivated a group of students at the Mobile and Portable Radio Research Group (MPRG) in Virginia Tech to create their own implementation of the SCA, which would later be called the Open-Source SCA Implementation :: Embedded, or OSSIE [12].

2.3.1 OSSIE and REDHAWK

OSSIE was the first open-source C++ implementation of the SCA. Created on a shoe-string budget and never tested for compliance, OSSIE still went on to become a valuable tool for students and researchers in the SDR community. Initially written to target Windows, and later ported to Linux, the OSSIE development team created new kinds of software radios targeting a multitude of embedded platforms [4] along with the earliest forms of SCA development tools [16]. The work by the OSSIE development team would eventually reach new heights in the REDHAWK [7] implementation.

REDHAWK is a complete re-write of OSSIE that builds upon the lessons learned from the students at MPRG. REDHAWK specifically targets Linux, and provides a complete IDE as an Eclipse plugin, along with many new development features, such as the novel Python "sandbox" for rapid component interaction and deployment. Additionally, REDHAWK allows developers to create components in C++, Java, and Python.

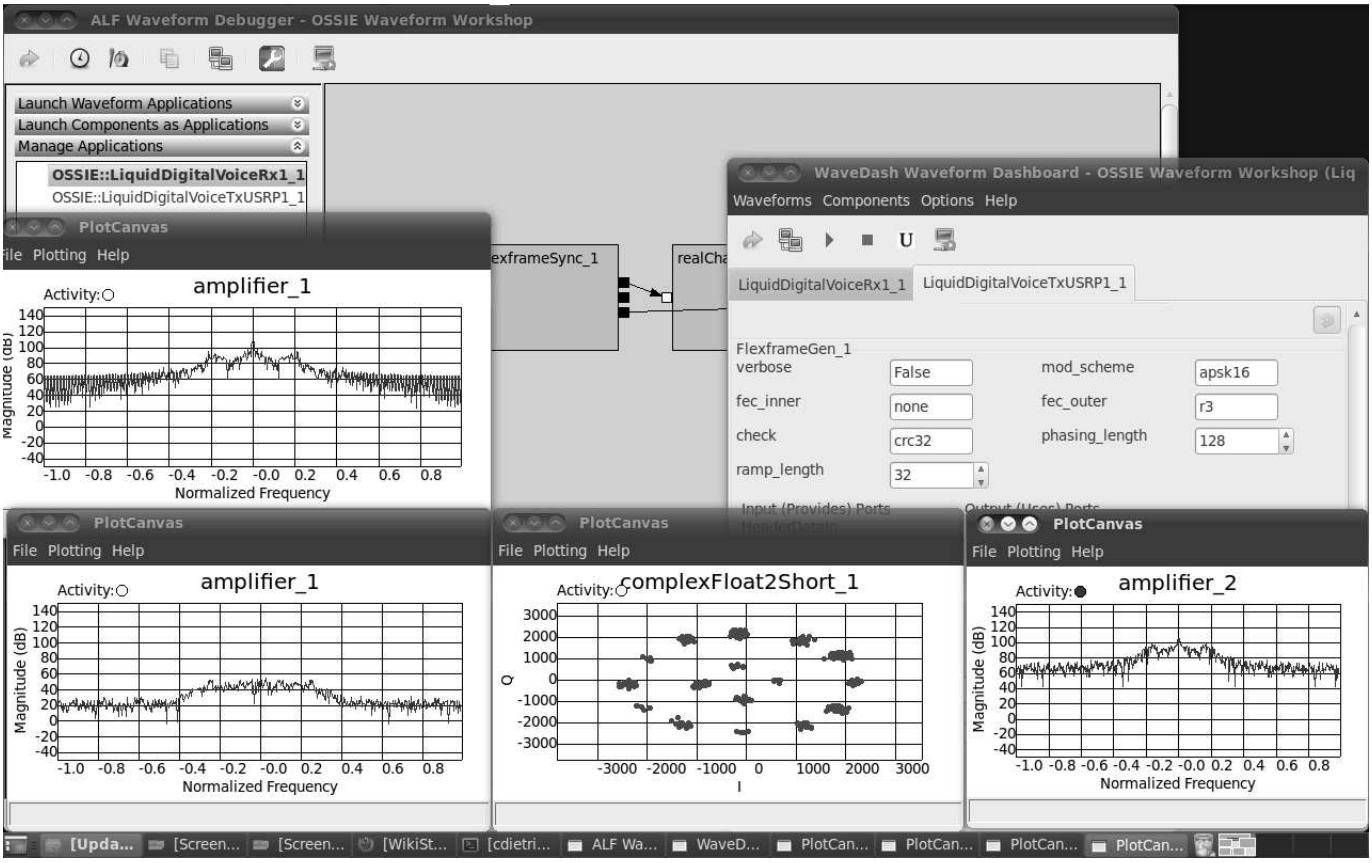

Figure 2.4: The OSSIE IDE

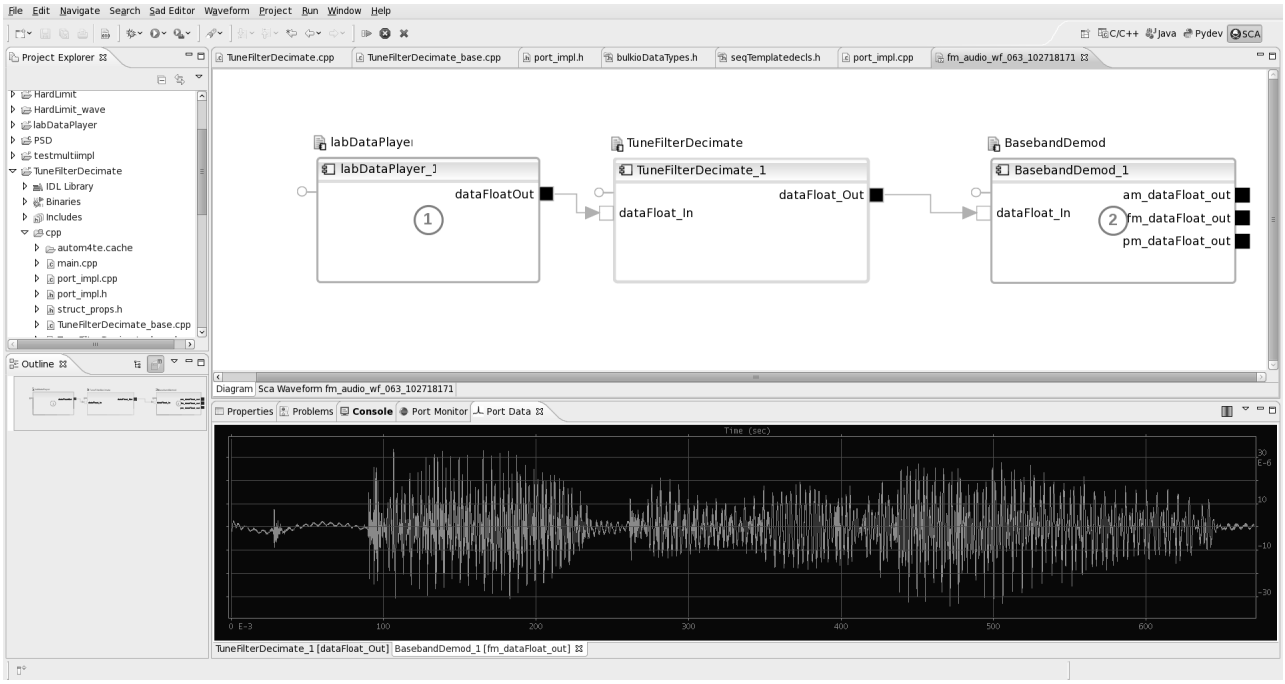

Figure 2.5: The REDHAWK IDE

Chapter 3

Testing Software by Fuzzing

3.1 Positive v Negative Testing

Positive testing is defined as providing the software under test with valid inputs and testing for valid output. Valid inputs and outputs are determined by the software specification. A simple example of positive testing might be an application that accepts the users age as input. Only unsigned integers would qualify as valid input. We would expect the system to respond with a message of success. If the response is not success, then the failure is recorded.

Negative testing is defined as providing the software under test with invalid inputs and testing for invalid output. Continuing with our example of accepting the users age as input, then testing with negative integers or characters would be invalid inputs. We would expect the system to respond with a message of error. If the response is not error, then the failure is recorded.

Our simple example of accepting the users age as input demonstrates that to test the software completely we must combine both positive and negative testing methods in order to cover all behavior associated with all possible inputs. The notion of coverage raises challenges for the software tester. First, it is not practical to test all possible inputs. Instead, it is preferable to test inputs at their boundaries, such as the maximum or minimum machine integer representation. Second, the set of test inputs should cover all paths of execution within the software under test. However, this requires a map of all execution paths and a mechanism to track the execution path while testing.

3.2 Types of Fuzzing

Software fuzzing is categorized into three types: Blackbox, Greybox, and Whitebox fuzzing. Blackbox fuzzing is defined by operating strictly on the inputs to a program, with no knowledge of the source code. An example of Blackbox fuzzing could be a text editor that opens a user specified file for editing. A Blackbox fuzzer would randomly generate a file, execute the text editor to open the generated file, then test the system for failure. Greybox fuzzing builds upon Blackbox fuzzing by introducing knowledge of the source code, typically the APIs. Knowledge of the API allows the software fuzzer to generate more relevant random tests that can cover all interfaces. For example, if it is known that a particular interface accepts an integer as a parameter, then it is more relevant to generate random integers versus random strings. Whitebox fuzzing builds upon Greybox fuzzing by providing complete knowledge of the source code, either in the form of source files or assembly instructions. Whitebox fuzzing is the premier software fuzzing method because it allows for greater code coverage by accounting for conditional statements while dramatically improving the relevance of each randomly generated test.

The relevance of each randomly generated test differs for each type of software fuzzer. Each type of software fuzzer treats the set of inputs as an undefined search space, and each successive type of software fuzzer is attempting to reduce the search space by leveraging knowledge of the source code to generate relevant tests.

3.3 Tools for Fuzzing

The majority of software fuzzing tools perform Blackbox fuzzing because most end users lack access to the underlying source code or APIs. In particular, the popular target of Blackbox fuzzers is web servers. In order to fuzz a web server, a model of the network protocol used by the web server is created and provided to the fuzz generator. The fuzz generator will randomize the parameters of the network protocol and send them to the web server. The web server is monitored by the fuzzing tool for any failures. If a failure is detected, then it is logged, and the fuzz generator continues to test the web server. The same mechanism that is used to model and fuzz a network protocol can be applied to file formats. For example, a model of an XML file can be created for fuzzing. Furthermore, the fuzzing generator can typically be extended to perform command line fuzzing.

The following is a listing of some of the more popular fuzzing tools that are currently under active development.

SPIKE [1]: a network protocol fuzzer written in C. The developer can choose to model a network protocol either as a configuration file to be provided to the Spike fuzz generator, or create a customized fuzzing tool using the Spike C API. Spike is widely used for its simplicity, but the code base is sparse and leaves the developer responsible for implementing additional

features, such as monitoring. Spike specifically targets Linux platforms.

PEACH [15]: a network protocol and file fuzzer written in C#. The developer can choose to model a network protocol or file format either as a configuration file, or can choose to extend the functionality of the tool using either C# or IronPython. Peach has gained in popularity because it is easy to learn and provides the developer with many additional features, including modeling state, system monitoring, and basic logging. Peach specifically targets Windows or OSX, but can also run in a Linux environment under 'mono'.

General Purpose Fuzzer (GPF) [9]: a network protocol fuzzer written in C. The GPF builds a description of the network protocol from a packet capture file provided by the user. This tool is novel in that it is designed to "learn" the network protocol and create a protocol description from real traffic. GPF specifically targets Linux, but it could feasibly be cross-compiled for alternative platforms.

KLEE [2]: a symbolic virtual machine built on top of the Low Level Virtual Machine (LLVM) that is capable of inspecting the disassembled instructions of a target executable binary and generating test cases that will attempt to completely cover all paths of execution. KLEE represents a form of Whitebox fuzzing and is a very advanced tool that has been used to identify failures and vulnerabilities in many common Linux utilities, such as Coreutils. While KLEE only supports command line fuzzing, it can be extended to provide support for modeling network protocols and file formats. KLEE specifically targets the Linux platform.

SAGE [8]: a state of the art Whitebox fuzzing tool developed at Microsoft Research Labs. Sage combines many different modeling and code analyses tools, similar to Peach and Klee. The distinguishing feature of Sage is the addition of a Constraint Solver that is continuously reducing the search space of the fuzz generator, allowing Sage to generate more relevant test cases and identify failures in less time. Sage only targets Windows, and is used exclusively by the Microsoft development teams.

Clearly there exists many tools for fuzzing across the spectrum from Blackbox to Whitebox. In order to develop a fuzzing tool for the SCA, we require the ability to define both network protocols and file formats. Based on this requirement, we can exclude Spike and GPF from the possible choices. The SAGE tool would be very useful, but it is available only for internal use by Microsoft developers. Only KLEE and Peach meet the requirements to define both network protocols and file formats. However, while KLEE provides immediate support for command line fuzzing, it does not provide immediate support for defining network protocols and file formats, and must be extended to meet the specific needs of the target application. Therefore, it is preferable to choose Peach over KLEE, because Peach provides immediate support for defining network protocols and file formats, while support for command line fuzzing would require creating an extension of the Peach framework.

In this paper, we will explore the capabilities of Peach, and present a set of test cases and the results of each test performed on the REDHAWK implementation of the Software Communications Architecture.

Chapter 4

Fuzzing with Peach

The Peach software fuzzing tool provides a foundation for modeling and fuzzing both network protocols and file formats. The foundation is built upon the definition of a configuration file in an XML format. The following sections discuss the anatomy of a Peach XML configuration file. For more detailed documentation of Peach, please visit http://www.peachfuzzer.com.

4.1 Peach Pit File

In Peach, a configuration file is referred to as a Peach Pit, or Pit file. A Pit file is composed of a model of the data format, a state model on how to act on the data model, an agent model on how to monitor the target application, and a test model on how to combine each of the former models. Each model in the Pit is described in XML and can be referenced by a unique name. The basic format of a Pit file is as follows:

Listing 4.1: Simple Example of a Peach Pit configuration file

```
<DataModel name=‘‘data_model1’’>
        <String name=‘‘data_model1_param’’ value=‘‘abcd’’ />
</DataModel>

<StateModel name=‘‘state_model1’’ initialState=‘‘init’’>
        <State name=‘‘init’’>
                <Action type=‘‘output’’>
                        <DataModel ref=‘‘data_model1’’ />
                </Action>
        </State>
</StateModel>

<Agent name=‘‘agent1’’>
        <Monitor class=‘‘LinuxDebugger’’ />
```

```
</Agent>

<Test name=‘‘Default’’>
        <StateModel ref=‘‘state_model1’’ />
        <Agent ref=‘‘agent1’’ />
        <Publisher type=‘‘File’’>
                <Param name=‘‘Filename’’ value=‘‘fuzz_output’’ />
        </Publisher>
</Test>
```

In the following sections, we will discuss each model type in the basic example above. Later, we will introduce the Pit files created to fuzz REDHAWK.

4.2 Peach DataModel Element

The Data Model is the primary element in Peach to create a model of a network protocol or data format. Peach Data Models are composed of child elements including String, Number, Block, and others. Each child element may be defined as mutable or immutable, which instructs the fuzz generator about which elements are constants and which are to be randomized. Additionally, Data Models may leverage a feature in Peach referred to as Analyzers. For example, Peach provides an XML Analyzer which is capable of reading a specified XML file and creating an internal DOM tree.

4.3 Peach StateModel Element

The State Model element describes how Peach will act on the Data Model with respect to the target application for each iteration of fuzzed data. Peach State Models are composed of one or more State child elements. Each State element defines one or more Action elements. An Action element can be of three types: Output, Input, Call, or user defined. An Output Action instructs Peach to output the fuzzed data to a defined Publisher (more on Publishers in the discussion on the Test element). An Input Action instructs Peach to accept data from a defined Publisher. A Call Action instructs Peach to interface with an Agent (more on Agents in the next section) to invoke a Monitor or a user defined process.

4.4 Peach Agent and Monitor Elements

The Agent element describes a Peach process that will fork and execute one or more Monitor processes. The Peach Agent process can be run either locally or remotely. The Monitor child

element describes a type of process monitor that Peach should invoke for the duration of the fuzzing run. Types of monitors include Process, Debugger, Crash, or user defined. The Process monitor can instruct Peach to monitor for an early exit. The Debugger monitor instructs Peach to attach a debugger to the target process. The Crash monitor will indicate a fault if a process fails and creates a core dump.

4.5 Peach Test Element

The Test element describes the composition of Agent and State Model references, along with the type Publishers to be used during a fuzzing run. The Publisher element instructs Peach where to pipe fuzzed data. Peach provides several Publisher types including Console, File, TCP, UDP, or user defined. Additionally, Peach provides a basic Logger element for logging the state of the fuzzing run. The Peach Logging utility is purposefully minimalist and is intended to be customized by the developer for their specific needs.

4.6 Peach Mutator and Strategy Element

The Peach Mutator Element defines how an element in a DataModel should be modified during a fuzzing run. Examples of a Mutator could be to generate random strings of length 1 to 10,000, or to generate random integers from INT_MIN to INT_MAX. Peach provides a number of default Mutator types. By default, Peach will attempt to employ every type of Mutator behavior. However, the developer can specify which Mutator to employ for a particular DataModel element. If the default Mutators are not appropriate for the application, then the developer may also choose to create their own Mutator types.

The Peach Strategy Element defines how a fuzzing run will be performed, such as modifying one DataModel element at a time, or modifying them all on each iteration. The developer can even specify a global Mutator to be applied to a Strategy. Furthermore, the developer has the freedom to create their own Strategy.

4.7 Peach Fixup Element

Peach is intended to provide the developer with a foundation for developing a customized fuzzing environment for their specific needs. To assist the developer in meeting this goal, Peach provides the Fixup element. Most Peach Pit elements support the Fixup child element. The Fixup element allows the developer to specify a script to be executed by Peach, in order to fix up a required feature. This is perhaps the most powerful feature of Peach because it allows the developer to extend the capabilities of Peach within the framework of Peach. For

example, the Logging feature of Peach is purposefully minimalist because it is expected that the developer will customize the logging for their specific needs. Furthermore, developers can create customized Agents, Monitors, Actions, Publishers, and Mutators. Currently, Peach provides support for Fixup scripts written in either C# or IronPython.

4.8 Running Peach

Running Peach is straight-forward once a Pit file has been created. At the prompt, simply type:

```
> peach your-pit-file.xml [test name]
```

and Peach will begin a fuzzing run. However, the premise of fuzzing is a brute force technique that will attempt every possible combination of inputs as defined by the Data Model. This means that a given fuzzing run can execute for an extremely extended period of time. In order to maximize the use of Peach, we can choose to define a finite set of tests to be performed as follows:

```
> peach -range 1,100 your-pit-file.xml
```

or we can leverage the built in multi-processor capabilities of Peach as follows:

```
> peach -p total-machines, this-machine-number your-pit-file.xml
  [test name]
> peach -p 10, 2 your-pit-file.xml [test name]
```

Even though Peach is generating random data, a given fuzzing run can be recreated by providing Peach with a seed value as follows:

```
> peach -seed seed-number your-pit-file.xml [test name]
> peach -seed 1000 your-pit-file.xml [test name]
```

If a Peach run is terminated early, or we want to start a Peach run from a particular run number, then Peach can be instructed to skip to a given test number as follows:

```
> peach -seed 1000 -skipto 100 your-pit-file.xml [test name]
```

To test a Pit file, Peach provides a debug mode that will run one iteration as follows:

```
> peach -1 -debug your-pit-file [test name]
```

Finally, Peach can be directed to perform a fuzzing run for a particular testing configuration as defined in a Pit file. If a test name is not given at the command line, then Peach will search the Pit file for the Test element with the name Default. However, additional tests may be defined by a unique name, and given to Peach at the command line.

Chapter 5

Fuzzing REDHAWK

The focus of this document is on Blackbox fuzzing of REDHAWK. As stated earlier, Blackbox fuzzing is a type of fuzzing that operates strictly on the inputs of the software with no knowledge of the underlying source code. Even though REDHAWK is open-source, we will operate only on the primary inputs. The primary inputs to REDHAWK are the XML Domain Profiles and CORBA. The Domain Profiles are critical to REDHAWK because they determine the configuration of the Core Framework (CF) to a known state, and they define the configuration of user defined software components. CORBA is the "Communications" in the SCA, and provides a middleware that abstracts the mechanisms of Inter-Process Communication (IPC) between software components.

For this case study in Blackbox fuzzing of REDHAWK, we will focus on the primary inputs to the CF, which are the XML Domain Profiles and CORBA. The CORBA specification is based on the General Inter-ORB Protocol (GIOP) v1.0, v1.1, and the latest v1.2. Various implementations of the CORBA specification exist both commercially and in the FOSS community. We specifically target FOSS implementations of CORBA because they are easily attainable and typically have a well documented history of improvement. REDHAWK uses omniORB, an open-source implementation of the CORBA specification. Meanwhile, the SCA specification is based on the definition of XML Domain Profiles and the CF. The CF is composed of different types of software components that manage the state of the framework. The primary management types in the CF are the CF::DomainManager, the CF::DeviceManager, the CF::ExecutableDevice, and the CF::ApplicationFactory. Each management type is assigned a set of XML Domain Profiles which define the initial configuration.

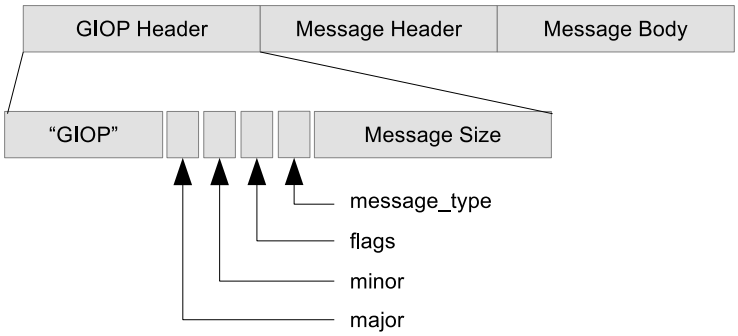

Figure 5.1: Format of the GIOP Header and Message

5.1 GIOP and the CORBA NameService

Fuzzing of the CORBA GIOP network protocol is an advanced use of Peach. CORBA acts as the software bus that allows all the components within the CF to communicate with one another. Creating a generalized DataModel of the GIOP specification is its own separate exploration. For the purposes of this case study, we chose to target the CORBA NameService, which acts as the router for all CORBA communications. Specifically, we create a DataModel of the "list" and "bind_new_context" commands supported by the NameService.

Every GIOP message begins with the GIOP Header format as shown in Figure 5.1. The GIOP Header is described as follows:

- The "GIOP" string identifies the protocol
- The GIOP version number, major and minor, is either 1.0, 1.1, or 1.2
- The flag byte indicates the endianess of the message
- The message type indicates if the message is a Request, Reply, or other
- The message size is the size of the message in bytes, excluding the GIOP Header

A GIOP Request message, as shown in Figure 5.2 describes the operation to be invoked on a particular instance of an object. The GIOP Request message is described as follows:

- The "service_contexts" field is used in conjunction with CORBA services to carry extra information along with the Request.
- The "request_id" field is a unique identifier used to match the Request with its associated Reply.

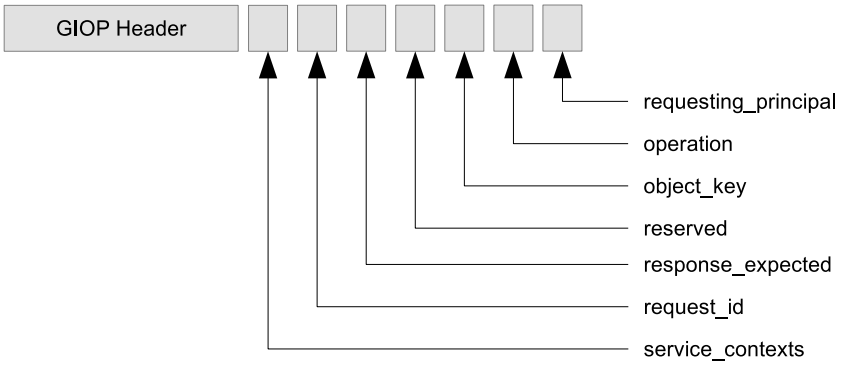

Figure 5.2: Format of the GIOP Request Message

- The "response_expected" flag indicates whether the Request is a one-way call or not, and is typically set as TRUE.
- Three bytes are reserved for future use.
- The "object_key" field indicates the object instance which will invoke the requested operation.
- The "operation" field is simply the name of the operation to be invoked.
- The "requesting_principal" field indicates the user making the request.

To create a DataModel of the "list" and "bind_new_context" commands that correctly adheres to the CORBA GIOP specification, we make use of the "nameclt" utility provided by omniORB and the Wireshark packet capture tool. The "nameclt" utility provides a command line interface to query and modify the CORBA NameService. By issuing the "list" and "bind_new_context" from the "nameclt" utility, we can use Wireshark to capture the resulting network traffic, as seen in Figure 5.3. The Wireshark packet captures, in combination with the CORBA GIOP specification, are used to create a DataModel of the "list" and "bind_new_context" commands in a Pit file.

To test the GIOP DataModel, all parameters of the DataModel are defined as immutable, and Peach is run in debug mode. This allows us to debug the DataModel until the results agree with the original Wireshark packet capture. Once a functional DataModel is created, we set all of the parameters of the DataModel as mutable, and execute Peach to fuzz the CORBA NameService.

```
0000  00 00 00 00 00 00 00 00  00 00 00 00 08 00 45 00   ........ ......E.
0010  00 98 f6 66 40 00 40 06  45 f7 7f 00 00 01 7f 00   ...f@.@. E.......
0020  00 01 84 2d 0a f9 d0 50  cc 4e 65 17 7a fa 80 18   ...-...P .Ne.z...
0030  01 01 fe 8c 00 00 01 01  08 0a 05 50 3a 29 05 50   ........ ...P:).P
0040  3a 29 47 49 4f 50 01 00  01 00 58 00 00 00 00 00   :)GIOP.. ..X.....
0050  00 00 02 00 00 00 01 00  00 00 0b 00 00 00 4e 61   ........ ......Na
0060  6d 65 53 65 72 76 69 63  65 00 06 00 00 00 5f 69   meServic e....._i
0070  73 5f 61 00 b2 00 00 00  00 00 28 00 00 00 49 44   s_a..... ..(...ID
0080  4c 3a 6f 6d 67 2e 6f 72  67 2f 43 6f 73 4e 61 6d   L:omg.or g/CosNam
0090  69 6e 67 2f 4e 61 6d 69  6e 67 43 6f 6e 74 65 78   ing/Nami ngContex
00a0  74 3a 31 2e 30 00                                 t:1.0.
```

Figure 5.3: GIOP Request packet capture

5.2 Core Framework and the XML Domain Profiles

Fuzzing of the XML Domain Profiles is straight-forward. A Pit file is created that defines a particular XML Domain Profile as the DataModel. We leverage the Peach XML Analyzer to simplify the DataModel. For this case study, we will be fuzzing the Properties Domain Profiles of the CF::DomainManager, the CF::DeviceManager, and the CF::ExecutableDevice. Additionally, we will be fuzzing the Software Assembly Domain Profile of the CF::ApplicationFactory.

The creation of a DataModel of the XML Domain Profiles is straight-forward in Peach. We simply leverage the Peach XML Analyzer to parse a specified XML Domain Profile and create a DOM tree for fuzzing. The basic template of the DataModel is as follows:

Listing 5.1: Peach DataModel using the XML Analyzer

```
<DataModel name=PRF>
  <String length=‘‘39’’ type=‘‘utf8’’ name=‘‘header’’ value=‘‘&lt;?xml
   version=&quot;1.0&quot; encoding=&quot;UTF8&quot;&gt;’’ />
        <String type=‘‘utf8’’ name=‘‘document’’>
                <Analyzer class=‘‘Xml’’ />
        </String>
</DataModel>
```

The DataModel above is easily reused for any type of XML Domain Profile. In this case study, we focus only on the CF management types and the Domain Profiles associated with them, as shown in Figure 5.4.

5.3 Test Setup

The fuzzing of REDHAWK v1.8.4 will be performed on a single CPU, such that all components within the REDHAWK Domain are colocated. The machine is an AMD Athlon II

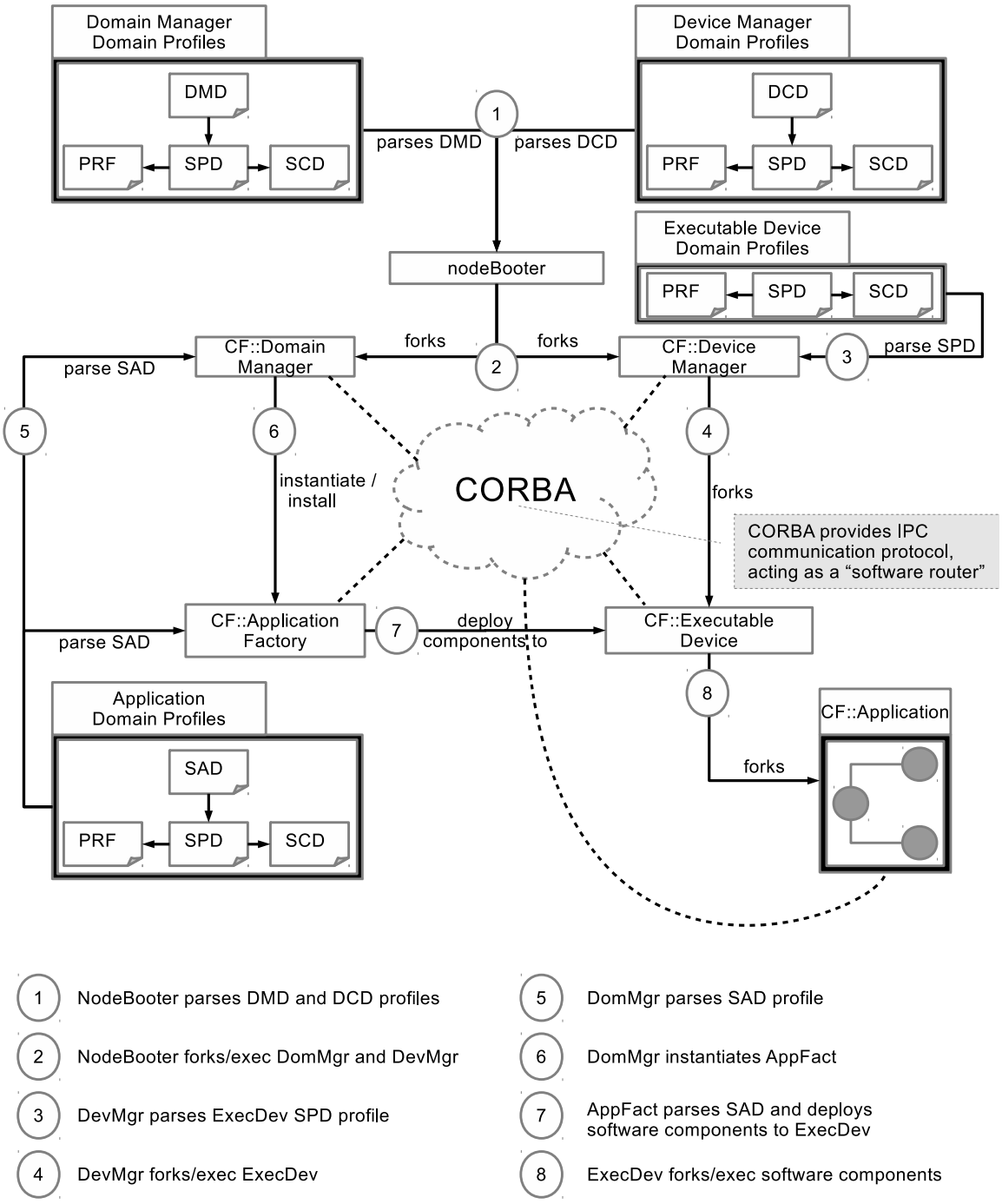

Figure 5.4: The startup sequence in the SCA.

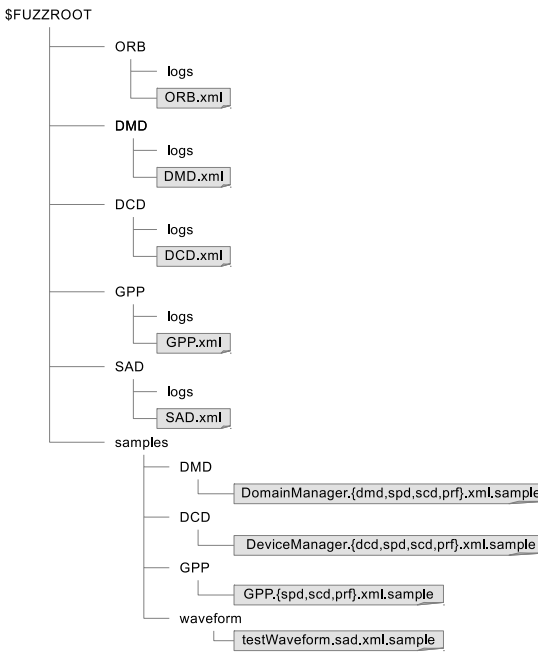

Figure 5.5: Directory Structure used during Fuzz Testing

Quad-Core with 8GB of RAM and is running Ubuntu v12.04 LTS. The following software versions are employed during the test:

- omniORB v4.1
- omniEvents v2.6.2
- REDHAWK v1.8.4
- Peach v3.0

With the exception of Peach, all software is installed from the Aptitude package manager. The Peach binaries can be downloaded and executed immediately.

A directory structure of each test is created to maintain organization of tests and results. Figure 5.5 shows the directory structure used during the tests.

Prior to running each fuzzing test, a set of commands are run from the *$FUZZROOT* location to guarantee that each test begins with identical conditions. First, the "NameService" and "EventService" are stopped.

```
$FUZZROOT> sudo service omniorb4-nameservice stop
$FUZZROOT> sudo service omniorb-eventservice stop
```

Then, any log files created by the “NameService” and “EventService” are removed. In the case of omniOrb, these log files maintain the last known state of the “NameService” and “EventService”. By removing them, we reset each service to their initial states.

```
$FUZZROOT> sudo rm /var/lib/omniorb/*
$FUZZROOT> sudo rm /var/lib/omniEvent/*
```

Finally, restart each service.

```
$FUZZROOT> sudo service omniorb4-nameservice restart
$FUZZROOT> sudo service omniorb-eventservice restart
```

Now each Peach fuzzing test will run with identical initial states. At the completion of a fuzzing run, any Domain Profiles that were modified are restored to their original state.

Chapter 6

Results of Fuzzing REDHAWK

Fuzzing of REDHAWK is performed on the CORBA NameService with a definition of the GIOP protocol, and the CF management components with a definition of their associated XML DomainProfiles. The CORBA NameService is fuzzed in a stand-alone mode, meaning that only the NameService is executed, and REDHAWK is not started. Furthermore, two implementations of the CORBA NameService are fuzzed identically, omniORB and JacORB. CF management components are fuzzed for each associated XML DomainProfile, with the exception of the CF::ApplicationFactory. For the CF::ApplicationFactory, only the SAD XML DomainProfile is fuzzed because this test occurs at the boundary of the CF and a user-defined application. Fuzzing the DomainProfiles associated with the SAD DomainProfile would be testing the user-defined application, and not the CF::ApplicationFactory. Fuzzing user-defined applications introduces a new set of challenges and is beyond the scope of this document.

6.1 Fuzzing the CORBA NameService

Fuzzing of the NameService "list" and "bind_new_context" commands against the omniORB CORBA implementation did not result in any failures. The omniORB NameService rejected most of the fuzzed commands. The times that the NameService did respond to a command was when the GIOP "reserved" parameter was fuzzed, which is treated as an optional parameter in GIOP.

As an additional test of the GIOP DataModel, the fuzzing tests were performed against an alternate CORBA implementation called JacORB. JacORB is a Java implementation of the CORBA specification. By simply adding another Test element to the Pit file, with no modifications to the DataModel, we successfully fuzzed the JacORB NameService using the same Pit file. Fuzzing of the JacORB NameService did not result in failure, but it did result in significant increases of memory utilization by the Java Virtual Machine (JVM),

which caused the operating system to thrash violently. The increased memory utilization is most likely due to an increased load on the Java Garbage Collector created by the excessive CORBA exceptions from the NameService rejecting too many commands. The omniORB implementation did not suffer a significant increase in memory utilization.

However, just because no failures were identified by this fuzzing test does not mean that either CORBA implementation is devoid of bugs. Both implementations are well established and the results of this test are a testament to their current strengths. Instead, a more intelligent strategy of fuzzing GIOP, beyond the naïve randomization of parameters, is needed to identify any failures in the implementations.

6.2 Fuzzing the Software Assembly Domain Profile

In this fuzzing test, the Software Assembly Domain Profile of the CF::ApplicationFactory is fuzzed. Fuzzing of the Software Assembly Domain Profile identified two failures in the CF. This fuzzing test can be recreated by running the following command:

```
$FUZZROOT> peach -seed 14022 --range 1,10 SAD/SAD.xml
```

The first failure is due to a race condition during the creation of a CF::Application by the CF::ApplicationFactory. It is possible to uninstall an application while it is being created. This prevents the application from later being released, resulting in one or more zombie processes. This failure was identified from the way that Peach executes a specified program. If given two or more executables to run, Peach will simply execute them all at once and let the operating system schedule their execution. However, we can instruct Peach to wait for a program to complete before executing the next. This is how the second failure was identified.

Listing 6.1: Fuzzed Software Assembly Domain Profile with Error

```
<componentfiles>
  <componentfile id="whitenoise_8b6df32b-ddd4-4cca-ba22-89585e2cfd6e"
  type="SPD">
    <!-- Additional component location information -->
  </componentfile>
</componentfiles>

<partitioning>
  <componentplacement>
    <componentfileref refid="AAAAAAAAAA" />
    <!-- Additional component information -->
  </componentplacement>
</partitioning>
```

The second failure is due to a malformed Software Assembly Domain Profile. The SAD profile defines the software components that make up a CF::Application. Each software component is given a unique identifier within the SAD profile. This unique identifier is later used within the SAD profile when matching a specific instance of the component to its implementation. If the matching fails, then the CF::DomainManager will still attempt to find the component implementation in the CF::FileSystem using a NULL pointer, causing a segmentation fault of the CF::DomainManager process.

Finding a software bug caused by a NULL pointer is a good example of the benefits of Blackbox Fuzzing as a software test tool. An edge case such as this can be easily overlooked by the development team.

6.3 Fuzzing the REDHAWK Management

In this fuzzing test, the Domain Profiles of the CF::DomainManager, CF::DeviceManager, and the CF::ExecutableDevice are fuzzed. By fuzzing the Domain Profiles, we are testing both the XML parsers and the configuration of the CF. In all cases, the XML parsers correctly identified a malformed Domain Profile. Similar to the results of fuzzing the CORBA NameService, a more intelligent strategy of fuzzing the XML Domain Profiles is needed to identify any failures in the parsers. However, while the XML parsers correctly identified malformed Domain Profiles, the CF exhibited erroneous behavior when attempting to shutdown the domain.

Starting with the CF::DomainManager, fuzzing tests were performed on the associated DMD, SPD, SCD, and PRF Domain Profiles. These tests can be reproduced with the following command:

```
$FUZZROOT> peach --range 1,100 DMD/DMD.xml DMD_{DMD,SPD,SCD,PRF}
```

Fuzz testing of the SCD did not produce any adverse effects, which suggests that the SCD is not used during the initial configuration of the CF::DomainManager. However, fuzz testing of the DMD, SPD, and PRF created multiple zombie processes of the CF::DomainManager and CF::DeviceManager. This is a critical error in REDHAWK because only one instance of the CF::DomainManager should exist within the domain. Moreover, the "NameService" retains an object reference to a CF::DomainManager, which suggests that the CF::DomainManager did not unbind its object reference when it attempted to exit. The combination of a zombie CF::DomainManager and CF::DeviceManager along with a lingering object reference to a CF::DomainManager means that the REDHAWK system domain is running in an invalid state but remains available for use.

The erroneous behavior found while fuzz testing the CF::DomainManager also occurs when fuzz testing the CF::DeviceManager. These tests can be reproduced with the following command:

```
$FUZZROOT> peach --range 1,100 DMD/DMD.xml DCD_{DCD,SPD,SCD,PRF_x86,
PRF_x86_64}
```

Fuzz testing of the SCD did not produce any adverse effects, just as in the fuzzing tests of the CF::DomainManager. However, fuzz testing of the PRF Domain Profiles resulted in the same erroneous behavior found when fuzz testing the CF::DomainManager. But fuzz testing of the DCD and SPD Domain Profiles resulted only in multiple zombie processes of the CF::DomainManager and not the CF::DeviceManager, which suggests that the CF::DeviceManager process correctly exits.

The same erroneous behavior found while fuzz testing the CF::DomainManager and CF::DeviceManager continues to manifest when fuzz testing the CF::ExecutableDevice. These test can be reproduced with the following command:

```
$FUZZROOT> peach --range 1,100 GPP/GPP.xml GPP_{SPD,SCD,PRF}
```

Fuzz testing of the CF::ExecutableDevice SPD, SCD, and PRF Domain Profiles displayed the same erroneous behavior as the previous tests. However, not only did these tests result in zombie CF::DomainManager and CF::DeviceManager processes, but also zombie CF::ExecutableDevice processes. A zombie process of the CF::ExecutableDevice represents a potentially serious vulnerability in REDHAWK. By vulnerability, we mean a failure that can be maliciously exploited. The purpose of the CF::ExecutableDevice is to fork/execute a given user-defined component. It is reasonable to speculate that a zombie CF::ExecutableDevice could be used to execute malicious code.

This erroneous behavior did not occur on every fuzz iteration, which suggests that this behavior could be attributed to a race condition when shutting down the CF. In particular, this race condition is created by running the CF with malformed input in rapid succession without any regard to synchronization. It could be argued that this behavior is unlikely to occur during the normal usage of the CF, but it demonstrates that it is possible to leave the CF in an invalid state and open to malicious use. By malicious use, we mean any use of the component for unintended purposes.

A zombie CF::ExecutableDevice is prone to malicious use. This particular component has the ability to fork and execute a child process with the same assigned permissions of the parent. It has already been shown how to recreate the GIOP protocol to execute commands, so it is not unreasonable to posit that a zombie CF::ExecutableDevice could be "spoofed" into accepting commands from a process pretending to be a CF::DeviceManager. Furthermore, it is reasonable to posit that a single process could mimic the entire CF simply by repeating the correct sequence of GIOP requests and responses. Therefore, a scenario exists whereby a malicious user could crash the CF with a malformed PRF, leaving behind a zombie CF::ExecutableDevice that can be "spoofed" into executing any other process requested of it.

Chapter 7

Conclusions, Recommendations, and Future Work

Software testing is a difficult task to perform. As a software project grows, its complexity increases making complete knowledge of the total system virtually impossible. Identifying and testing for edge cases in large software projects will always be an incomplete task and no singular method exists to address this challenge. Blackbox fuzzing provides a methodology for initial system testing. Moreover, a good blackbox fuzzer can be extended to incorporate new forms of testing.

The use of Peach v3.0 framework proved to be beneficial towards the creation of a customized fuzzing tool for REDHAWK. In particular, the documentation available is sufficient to provide guidance during the development process. Running Peach in a Linux environment requires installation of the complete Mono run-time environment, including developer libraries. In order to extend Peach, installation of IronPython is required. During development of Peach Pit definition files, the use of the Peach debug feature helps to quickly validate the initial behavior of a fuzz test.

The error found during fuzz testing of the Software Assembly Domain Profile is a good example of the benefits of blackbox fuzzing. This error represents an edge case that was overlooked by the development team, but that can easily be resolved as a bug fix. Additionally, the test case that identified the error can be replicated to test a proposed bug fix.

The erroneous behavior of zombie processes found during fuzz testing of the CF management components represents a larger problem within the implementation of REDHAWK. It should be noted that the challenge of fault detection is not limited to systems employing CORBA, but any distributed multi-process system. Fault detection in distributed multi-process systems is an active area of research with many proposed solutions, each unique to the application space.

When a fault occurs within REDHAWK, the challenge becomes the proper synchronization

of the error handlers across multiple, and even distributed, processes. There are several possible solutions to resolve the erroneous behavior of zombie processes within REDHAWK. The best place to begin is in the synchronization of collocated components. It would be beneficial to allocate a segment of shared memory to share component status information across processes within the domain. Status information would contain the current state of a component such as initializing, configuring, running, exiting, and error states. This would be similar to a status register found in a microcontroller. Synchronization of access to this shared memory can be accomplished through named mutexes and conditions. At a minimum, the CF::DomainManager would have ownership of a "DomMgr" mutex to achieve the singleton behavior required of the domain, and a second "status" mutex would be used by all components to control access to the shared memory space. Named conditions can be used to atomically signal one or more processes about an event, such as a particular error.

The use of named mutexes and conditions is only sufficient to achieve synchronization of collocated components. If the domain components are distributed across a network, then the challenge of fault detection, notification, and synchronization is significantly harder. The CORBA specification provides what is called the Portable Interceptor, which is a hook for developers to "listen" to the communication traffic of a particular component. It would be beneficial to provide a domain Monitor that has hooks in every component within the domain, and "listening" for errors and coordinating any required error handling across the domain. However, the design and implementation of distributed fault detection, notification, and synchronization of error handling is a very involved process. As stated earlier, fault detection for distributed multi-process systems is an active area of research. The OMG created the CORBA Fault Tolerance (FT) [13] specification to address the challenges of designing and implementing distributed component Monitors. The FT specification expands on the CORBA GIOP specification by introducing fault parameters within the protocol along with a new FT network service to address fault detection and notification. Unfortunately, most implementations of CORBA do not support the FT specification. The ACE/TAO CORBA implementation is the only FOSS resource that provides support for the FT specification.

In order to sufficiently address the erroneous behavior of zombie processes within REDHAWK, it is necessary to implement all of the aforementioned recommendations. Employment of shared memory component state information along with named mutexes and conditions for synchronization of collocated domain components will address the problem locally. An implementation of a domain Monitor that leverages the CORBA Portable Interceptor to "listen" for errors across domain components and provide synchronization of error handling will provide a basic mechanism for distributed fault detection. Ultimately, the best solution for distributed application spaces is for REDHAWK to provide support for the ACE/TAO CORBA implementation, in order to leverage support for the FT service.

Blackbox fuzzing of the REDHAWK SCA has identified both a software bug and a systemic flaw within the CF. However, the fuzz tests performed in this case study should be expanded and customized for REDHAWK. For instance, a customized Monitor that examines the state of the domain on each iteration would be useful. Additionally, custom Mutator types that

address the specific ways that the CF parses and uses the Domain Profiles would allow deeper testing of the parsers and components. In time, a customized blackbox fuzzer of REDHAWK will have to run longer to identify any failures. At that point, it will be beneficial to expand the fuzzing tests to incorporate feedback from a static code analyzer in order to improve the relevance and coverage of the fuzzing tests.

Bibliography

[1] Immunity: Knowing you're secure. `http://www.immunitysec.com/downloads/SPIKE2.9.tgz`.

[2] The KLEE Symbolic Virtual Machine. `http://klee.github.io/klee/`.

[3] Abbas Abbaspour-Tamijani, L. Dussopt, and G.M. Rebeiz. Miniature and tunable filters using mems capacitors. *Microwave Theory and Techniques, IEEE Transactions on*, 51(7):1878–1885, July 2003.

[4] Carlos Roberto Aguayo Gonzalez. Design and implementation of an efficient sca framework for software-defined radios. Master's thesis, Virginia Polytechnic Institute and State University, May 2006.

[5] D. Cabric, S.M. Mishra, and R.W. Brodersen. Implementation issues in spectrum sensing for cognitive radios. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, volume 1, pages 772–776 Vol.1, Nov 2004.

[6] Joint Tactical Networking Center. JTRS Standards for Software Communications Architecture (SCA). `http://jtnc.mil/sca/Pages/sca1.aspx`.

[7] GEONTECH. `http://redhawksdr.org`.

[8] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. *Queue*, 10(1):20:20–20:27, January 2012.

[9] VDA Labs. `http://www.vdalabs.com/tools/efs_gpf.html`.

[10] R.I Lackey and D.W. Upmal. Speakeasy: the military software radio. *Communications Magazine, IEEE*, 33(5):56–61, May 1995.

[11] J. Mitola. The software radio architecture. *Communications Magazine, IEEE*, 33(5):26–38, May 1995.

[12] Mobile, Portable Radio Research Group (MPRG) at Virginia Polytechnic Institute, and State University (VPI&SU). OSSIE — SCA-Based Open Source Software Defined Radio. `http://ossie.wireless.vt.edu/`.

[13] Object Management Group (OMG). FT. `http://www.omg.org/spec/FT/1.0/`, May 2010.

[14] Object Management Group (OMG). CORBA 3.3. `http://www.omg.org/spec/CORBA/3.3/`, November 2012.

[15] Deja Vu Security. Peach 3. `http://peachfuzzer.com`.

[16] Jason Snyder, Deepan Seeralan, Shereef Sayed, Jeffery Wilson, Carl B. Dietrich, Stephen H. Edwards, and Jeffrey H. Reed. Open source software-defined radio tools for education, research, and rapid prototyping. *International Journal on Software Tools for Technology Transfer*, 16(1):67–80, 2014.

[17] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[18] Zhi Tian and G.B. Giannakis. Compressed sensing for wideband cognitive radios. In *Acoustics, Speech and Signal Processing, 2007. ICASSP 2007. IEEE International Conference on*, volume 4, pages IV–1357–IV–1360, April 2007.

Appendix A

DomainManager Pit File

Listing A.1: DomainManager Pit File

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://peachfuzzer.com/2012/Peach
    /peach/peach.xsd">

    <!-- BEGIN DMD DataModel and StateModel -->
    <DataModel name="DMD">
        <String length="39" type="utf8" name="header"
            value="&lt;?xml version=&quot;1.0&quot;
            encoding=&quot;UTF-8&quot;?&gt;" />
        <String length="112" type="utf8" name="doctype"
            value="&lt;!DOCTYPE domainmanagerconfiguration
            PUBLIC &quot;-//JTRS//DTD SCA V2.2.2 DMD//EN&quot;
            &quot;domainmanagerconfiguration.dtd&quot;&gt;" />
        <String type="utf8" name="document" >
            <Analyzer class="Xml" />
        </String>
    </DataModel>

    <StateModel name="DMDState" initialState="initial">
        <State name="initial">
            <Action type="output">
                <DataModel ref="DMD" />
                <Data fileName="samples/DMD/DomainManager.dmd.xml.sample" />
            </Action>
            <Action type="close" />
        </State>
    </StateModel>
```

```
<!-- END DMD DataModel and StateModel -->

<!-- BEGIN PRF DataModel and StateModel -->
<DataModel name="PRF">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="80" type="utf8" name="doctype"
        value="&lt;!DOCTYPE properties PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 PRF//EN&quot;
        &quot;properties.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="PRFState" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="PRF" />
            <Data fileName="samples/DMD/DomainManager.prf.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END PRF DataModel and StateModel -->

<!-- BEGIN SPD DataModel and StateModel -->
<DataModel name="SPD">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="74" type="utf8" name="doctype"
        value="&lt;!DOCTYPE softpkg PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 SPD//EN&quot;
        &quot;softpkg.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="SPDState" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="SPD" />
```

```
            <Data fileName="samples/DMD/DomainManager.spd.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END SPD DataModel and StateModel -->

<!-- BEGIN SCD DataModel and StateModel -->
<DataModel name="SCD">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="94" type="utf8" name="doctype"
        value="&lt;!DOCTYPE softwarecomponent PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 SCD//EN&quot;
        &quot;softwarecomponent.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="SCDState" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="SCD" />
            <Data fileName="samples/DMD/DomainManager.scd.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END SCD DataModel and StateModel -->

<Agent name="LinAgent">
    <Monitor class="Process">
        <Param name="Executable" value="nodeBooter" />
        <Param name="Arguments"
            value="-D -d /nodes/DevMgr_rfrtxs/DeviceManager.dcd.xml
            -debug 2 --nopersist --force-rebind" />
        <Param name="RestartOnEachTest" value="true" />
        <Param name="FaultOnEarlyExit" value="true" />
    </Monitor>
</Agent>

<!-- BEGIN Test DMD PRF -->
<Test name="DMD_PRF">
```

```
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="PRFState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dom/mgr/DomainManager.prf.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DMD PRF -->

<!-- BEGIN Test DMD SPD -->
<Test name="DMD_SPD">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="SPDState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dom/mgr/DomainManager.spd.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DMD SPD -->

<!-- BEGIN Test DMD SCD -->
<Test name="DMD_SCD">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="SCDState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dom/mgr/DomainManager.scd.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DMD SCD -->

<!-- BEGIN Test DMD -->
<Test name="DMD_DMD">
```

```
        <Agent ref="LinAgent" platform="linux" />
        <StateModel ref="DMDState" />
        <Publisher class="File">
            <Param name="FileName"
                value="/var/redhawk/sdr/dom/domain/DomainManager.dmd.xml" />
        </Publisher>
        <Logger class="File">
            <Param name="Path" value="logs" />
        </Logger>
        <!-- <Publisher class="Console" /> -->
    </Test>
    <!-- END Test DMD -->
</Peach>
<!-- end -->
```

Appendix B

DeviceManager Pit File

Listing B.1: DeviceManager Pit File

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://peachfuzzer.com/2012/Peach
    /peach/peach.xsd">

    <!-- BEGIN DCD DataModel and StateModel -->
    <DataModel name="DCD">
        <String length="39" type="utf8" name="header"
            value="&lt;?xml version=&quot;1.0&quot;
            encoding=&quot;UTF-8&quot;?&gt;" />
        <!--
        <String length="80" type="utf8" name="doctype"
        value="&lt;!DOCTYPE properties PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 PRF//EN&quot;
        &quot;properties.dtd&quot;&gt;" />
        -->
        <String type="utf8" name="document" >
            <Analyzer class="Xml" />
        </String>
    </DataModel>

    <StateModel name="DCDState" initialState="initial">
        <State name="initial">
            <Action type="output">
                <DataModel ref="DCD" />
                <Data fileName="samples/DCD/DeviceManager.dcd.xml.sample" />
            </Action>
            <Action type="close" />
```

```
    </State>
</StateModel>
<!-- END DCD DataModel and StateModel -->

<!-- BEGIN SPD DataModel and StateModel -->
<DataModel name="SPD">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="74" type="utf8" name="doctype"
        value="&lt;!DOCTYPE softpkg PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 SPD//EN&quot;
        &quot;softpkg.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="SPDState" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="SPD" />
            <Data fileName="samples/DCD/DeviceManager.spd.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END SPD DataModel and StateModel -->

<!-- BEGIN SCD DataModel and StateModel -->
<DataModel name="SCD">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="94" type="utf8" name="doctype"
        value="&lt;!DOCTYPE softwarecomponent PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 SCD//EN&quot;
        &quot;softwarecomponent.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="SCDState" initialState="initial">
    <State name="initial">
```

```
        <Action type="output">
            <DataModel ref="SCD" />
            <Data fileName="samples/DCD/DeviceManager.scd.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END SCD DataModel and StateModel -->

<!-- BEGIN PRF x86 DataModel and StateModel -->
<DataModel name="PRF_x86">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="80" type="utf8" name="doctype"
        value="&lt;!DOCTYPE properties PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 PRF//EN&quot;
        &quot;properties.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="PRFState_x86" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="PRF_x86" />
            <Data fileName="samples/DCD/
                DeviceManager.Linux.x86.prf.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END PRF x86 DataModel and StateModel -->

<!-- BEGIN PRF x86_64 StateModel -->
<StateModel name="PRFState_x86_64" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="PRF_x86" />
            <Data fileName="samples/DCD/
                DeviceManager.Linux.x86_64.prf.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
```

```
</StateModel>
<!-- END PRF x86 DataModel and StateModel -->

<Agent name="LinAgent">
    <Monitor class="Process">
        <Param name="Executable" value="nodeBooter" />
        <Param name="Arguments"
            value="-D -d /nodes/DevMgr_rfrtxs/DeviceManager.dcd.xml
            -debug 2 --nopersist --force-rebind" />
        <Param name="RestartOnEachTest" value="true" />
        <Param name="FaultOnEarlyExit" value="true" />
    </Monitor>
</Agent>

<!-- BEGIN Test DCD -->
<Test name="DCD_DCD">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="DCDState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/nodes/DevMgr_rfrtxs/
            DeviceManager.dcd.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="./DCD/logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DCD -->

<!-- BEGIN Test DCD SPD -->
<Test name="DCD_SPD">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="SPDState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/mgr/DeviceManager.spd.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="./DCD/logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DCD SPD -->
```

```
<!-- BEGIN Test DCD SCD -->
<Test name="DCD_SCD">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="SCDState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/mgr/DeviceManager.scd.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="./DCD/logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DCD SCD -->

<!-- BEGIN Test DCD PRF x86 -->
<Test name="DCD_PRF_x86">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="PRFState_x86" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/mgr/
            DeviceManager.Linux.x86.prf.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="./DCD/logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test DCD PRF x86 -->

<!-- BEGIN Test DCD PRF x86_64 -->
<Test name="DCD_PRF_x86_64">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="PRFState_x86_64" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/mgr/
            DeviceManager.Linux.x86_64.prf.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="./DCD/logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
```

```
    <!-- END Test DCD PRF x86_64 -->
</Peach>
<!-- end -->
```

Appendix C

ExecutableDevice Pit File

Listing C.1: ExecutableDevice Pit File

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://peachfuzzer.com/2012/Peach
    /peach/peach.xsd">

    <!-- BEGIN SPD DataModel and StateModel -->
    <DataModel name="SPD">
        <String length="39" type="utf8" name="header"
            value="&lt;?xml version=&quot;1.0&quot;
            encoding=&quot;UTF-8&quot;?&gt;" />
        <!--
        <String length="80" type="utf8" name="doctype"
        value="&lt;!DOCTYPE properties PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 PRF//EN&quot;
        &quot;properties.dtd&quot;&gt;" />
        -->
        <String type="utf8" name="document" >
            <Analyzer class="Xml" />
        </String>
    </DataModel>

    <StateModel name="SPDState" initialState="initial">
        <State name="initial">
            <Action type="output">
                <DataModel ref="SPD" />
                <Data fileName="samples/GPP/GPP.spd.xml.sample" />
            </Action>
            <Action type="close" />
```

```
    </State>
</StateModel>
<!-- END SPD DataModel and StateModel -->

<!-- BEGIN PRF DataModel and StateModel -->
<DataModel name="PRF">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <!--
    <String length="80" type="utf8" name="doctype"
    value="&lt;!DOCTYPE properties PUBLIC
    &quot;-//JTRS//DTD SCA V2.2.2 PRF//EN&quot;
    &quot;properties.dtd&quot;&gt;" />
    -->
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>

<StateModel name="PRFState" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="PRF" />
            <Data fileName="samples/GPP/GPP.prf.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END PRF DataModel and StateModel -->

<!-- BEGIN SCD DataModel and StateModel -->
<DataModel name="SCD">
    <String length="39" type="utf8" name="header"
        value="&lt;?xml version=&quot;1.0&quot;
        encoding=&quot;UTF-8&quot;?&gt;" />
    <String length="94" type="utf8" name="doctype"
        value="&lt;!DOCTYPE softwarecomponent PUBLIC
        &quot;-//JTRS//DTD SCA V2.2.2 SCD//EN&quot;
        &quot;softwarecomponent.dtd&quot;&gt;" />
    <String type="utf8" name="document" >
        <Analyzer class="Xml" />
    </String>
</DataModel>
```

```
<StateModel name="SCDState" initialState="initial">
    <State name="initial">
        <Action type="output">
            <DataModel ref="SCD" />
            <Data fileName="samples/GPP/GPP.scd.xml.sample" />
        </Action>
        <Action type="close" />
    </State>
</StateModel>
<!-- END PRF DataModel and StateModel -->

<Agent name="LinAgent">
    <Monitor class="Process">
        <Param name="Executable" value="nodeBooter" />
        <Param name="Arguments"
            value="-D -d /nodes/DevMgr_rfrtxs/DeviceManager.dcd.xml
            -debug 2" />
        <Param name="RestartOnEachTest" value="true" />
        <Param name="FaultOnEarlyExit" value="true" />
    </Monitor>
</Agent>

<!-- BEGIN Test GPP SPD -->
<Test name="GPP_SPD">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="SPDState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/devices/GPP/GPP.spd.xml" />
    </Publisher>
    <Logger class="File">
        <Param name="Path" value="./GPP/logs" />
    </Logger>
    <!-- <Publisher class="Console" /> -->
</Test>
<!-- END Test GPP SPD -->

<!-- BEGIN Test GPP PRF -->
<Test name="GPP_PRF">
    <Agent ref="LinAgent" platform="linux" />
    <StateModel ref="PRFState" />
    <Publisher class="File">
        <Param name="FileName"
            value="/var/redhawk/sdr/dev/devices/GPP/GPP.prf.xml" />
    </Publisher>
```

```
        <Logger class="File">
            <Param name="Path" value="./GPP/logs" />
        </Logger>
        <!-- <Publisher class="Console" /> -->
    </Test>
    <!-- END Test GPP PRF -->

    <!-- BEGIN Test GPP SCD -->
    <Test name="GPP_SCD">
        <Agent ref="LinAgent" platform="linux" />
        <StateModel ref="SCDState" />
        <Publisher class="File">
            <Param name="FileName"
                value="/var/redhawk/sdr/dev/devices/GPP/GPP.scd.xml" />
        </Publisher>
        <Logger class="File">
            <Param name="Path" value="./GPP/logs" />
        </Logger>
        <!-- <Publisher class="Console" /> -->
    </Test>
    <!-- END Test GPP SCD -->
</Peach>
<!-- end -->
```

Appendix D

ApplicationFactory Pit File

Listing D.1: ApplicationFactory Pit File

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://peachfuzzer.com/2012/Peach
    /peach/peach.xsd">

    <!-- BEGIN SAD DataModel and StateModel -->
    <DataModel name="SAD">
        <String length="39" type="utf8" name="header"
            value="&lt;?xml version=&quot;1.0&quot;
            encoding=&quot;UTF-8&quot;?&gt;" />
        <String length="92" type="utf8" name="doctype"
            value="&lt;!DOCTYPE properties PUBLIC
            &quot;-//JTRS//DTD SCA V2.2.2 PRF//EN&quot;
            &quot;softwareassembly.dtd&quot;&gt;" />
        <String type="utf8" name="document" >
            <Analyzer class="Xml" />
        </String>
    </DataModel>

    <StateModel name="SADState" initialState="fuzzWaveform">
        <State name="fuzzWaveform">
            <Action type="output">
                <DataModel ref="SAD" />
                <Data fileName="samples/waveform/
                    testWaveform.sad.xml.sample" />
            </Action>
            <Action type="close" />
            <Action type="changeState" ref="installSCAApp" />
```

```
    </State>
    <State name="installSCAApp">
        <Action type="call" method="installApp"
            publisher="Peach.Agent" />
        <Action type="changeState" ref="createSCAApp" />
    </State>
    <State name="createSCAApp">
        <Action type="call" method="createApp"
            publisher="Peach.Agent" />
        <Action type="changeState" ref="releaseSCAApp" />
    </State>
    <State name="releaseSCAApp">
        <Action type="call" method="releaseApp"
            publisher="Peach.Agent" />
        <Action type="changeState" ref="uninstallSCAApp" />
    </State>
    <State name="uninstallSCAApp">
        <Action type="call" method="uninstallApp"
            publisher="Peach.Agent" />
    </State>
</StateModel>
<!-- END SAD DataModel and StateModel -->

<Agent name="LinAgent">
    <Monitor class="Process">
        <Param name="Executable" value="scaclt" />
        <Param name="Arguments"
            value="install REDHAWK_DEV
            /waveforms/testWaveform/testWaveform.sad.xml" />
        <Param name="WaitForExitOnCall" value="installApp" />
    </Monitor>
    <Monitor class="Process">
        <Param name="Executable" value="scaclt" />
        <Param name="Arguments"
            value="create REDHAWK_DEV
            DCE:86f5c2f3-0b60-49b6-89f2-d6b73fdf7f0c testWaveform_1" />
        <Param name="WaitForExitOnCall" value="createApp" />
    </Monitor>
    <Monitor class="Process">
        <Param name="Executable" value="scaclt" />
        <Param name="Arguments"
            value="release REDHAWK_DEV
            DCE:86f5c2f3-0b60-49b6-89f2-d6b73fdf7f0c" />
        <Param name="WaitForExitOnCall" value="releaseApp" />
    </Monitor>
```

```
        <Monitor class="Process">
            <Param name="Executable" value="scaclt" />
            <Param name="Arguments"
                value="uninstall REDHAWK_DEV
                DCE:86f5c2f3-0b60-49b6-89f2-d6b73fdf7f0c" />
            <Param name="WaitForExitOnCall" value="uninstallApp" />
        </Monitor>
    </Agent>

    <!-- BEGIN Test SAD -->
    <Test name="Default">
        <Agent ref="LinAgent" platform="linux" />
        <StateModel ref="SADState" />
        <Publisher class="File">
            <Param name="FileName"
                value="/var/redhawk/sdr/dom/waveforms/testWaveform/
                testWaveform.sad.xml" />
        </Publisher>
        <Logger class="File">
            <Param name="Path" value="logs" />
        </Logger>
        <!-- <Publisher class="Console" /> -->
    </Test>
    <!-- END Test SAD -->

</Peach>
<!-- end -->
```

Appendix E

GIOP “list” operation Pit File

Listing E.1: GIOP “list” operation Pit File

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://peachfuzzer.com/2012/Peach
    /peach/peach.xsd">

    <!--
         File: GIOP_list.xml
         Description: Provides a model of the CORBA GIOP v1.0 protocol.
         The protocol models the 'list' command provided by the CORBA
         NameService, which lists the named contexts and objects currently
         bound to the NameService. All parameters of the protocol are
         defined as 'mutable', meaning that all parameters are eligible to
         be fuzzed by Peach.
    -->

    <!-- Model of GIOP Header v1.0 -->
    <DataModel name="GIOPHeader_1_0">
        <String name="magic" length="4" value="GIOP" />
        <Number name="major" size="8" signed="false" value="1" />
        <Number name="minor" size="8" signed="false" value="0" />
        <Number name="byte_order" size="8" signed="false" value="1" />
        <Number name="message_type" size="8" signed="false" />
        <Number name="message_size" size="32" signed="false" />
    </DataModel>

    <!-- Model of a GIOP Request v1.0 -->
    <DataModel name="GIOPRequest_1_0" ref="GIOPHeader_1_0">
        <!-- message type = 0 indicates a GIOP Request -->
```

```
<Number name="message_type" size="8" signed="false" value="0" />
<!-- message size is sizeof(Request Header)+sizeof(Request Body) -->
<Number name="message_size" size="32" signed="false">
    <Relation type="size" of="Request_1_0" />
</Number>
<!-- Request Header -->
<Block name="Request_1_0">
    <!-- service context is always 0, typically -->
    <Number name="service_context" size="32" signed="false"
        value="0" />
    <!-- request id is mirrored by the reply -->
    <Number name="request_id" size="32" signed="false"
        value="2" />
    <!-- response = 1 indicates a reply is expected -->
    <Number name="response_expected" size="8" signed="false"
        value="1"/>
    <!-- 3 bytes reserved for future use -->
    <Number name="reserved" size="24" signed="false" value="0" />
    <Number name="object_key_length" size="32" signed="false">
        <Relation type="size" of="object_key" />
    </Number>
    <String name="object_key" value="NameService"
        nullTerminated="false" />
    <Number name="null1" size="8" signed="false" value="0" />
    <Number name="operation_length" size="32" signed="false">
        <Relation type="size" of="operation" />
    </Number>
    <!-- initial request operation is usually '_is_a' -->
    <String name="operation" value="_is_a" nullTerminated="true" />
    <!-- in CORBA, evrey method takes at least one parameter -->
    <!-- the byte code of operation parameters is define by TcKind
         mapping in the OMG CORBA specification -->
    <Number name="operation_parameters" value="b200" valueType="hex"
        size="16" signed="false" />
    <!-- technically, this parameter is deprecated and is
         typically 0 -->
    <Number name="requesting_principal_length" size="32"
        signed="false" value="0" />
    <!-- Request Body -->
    <Block name="RequestBody">
        <!-- parameters of the Request Body change based on the
             requested object key and operation -->
        <!-- these parameters are set for the '_is_a' operation -->
        <Number name="type_id_length" size="32" signed="false">
            <Relation type="size" of="type_id" />
```

```
            </Number>
            <!-- type id describing the object mapping -->
            <String name="type_id"
                value="IDL:omg.org/CosNaming/NamingContext:1.0"
                nullTerminated="true" />
        </Block>
    </Block>
</DataModel>

<!-- override the base request to model the 'list' operation -->
<DataModel name="GIOPRequest_1_0_list" ref="GIOPRequest_1_0">
    <!-- override the request operation -->
    <String name="Request_1_0.operation" value="list"
        nullTerminated="true" />
    <!-- override the request operation parameters -->
    <Number name="Request_1_0.operation_parameters" value="00b200"
        valueType="hex" size="24" signed="false" />
    <!-- override the request body type id -->
    <String name="Request_1_0.RequestBody.type_id" value=""
        length="0" />
</DataModel>

<!-- model of a generic reply from the NameService -->
<DataModel name="GIOPReply">
    <!-- since we are modelling the client side, we are currently
         ignoring the specifics of the server response -->
    <String name="reply" value="" />
</DataModel>

    <StateModel name="stateRequest_1_0" initialState="Initial">
    <State name="Initial">
        <!--
        <Action type="output">
            send initial GIOP Request
            <DataModel ref="GIOPRequest_1_0" />
        </Action>
        <Action type="input">
            recieve whatever the server responds with
            <DataModel ref="GIOPReply" />
        </Action>
        -->
        <Action type="output">
            <!-- send the fuzzed 'list' command -->
            <DataModel ref="GIOPRequest_1_0_list" />
        </Action>
```

```
        <Action type="input">
            <!-- recieve whatever the server responds with -->
            <DataModel ref="GIOPReply" />
        </Action>
        <!-- Rinse and repeat -->
    </State>
</StateModel>

<!-- define our peach agent and process monitors -->
<Agent name="omniAgent">
    <!--
    <Monitor class="Pcap">
        <Param name="Device" value="lo" />
        <Param name="Filter" value="port 2809" />
    </Monitor>
    -->
    <Monitor class="Process">
        <Param name="Executable" value="omniNames" />
        <Param name="Arguments"
            value="-logdir ./logs -errlog ./logs/err.log" />
        <!--
        <Param name="RestartOnEachTest" value="true" />
        -->
        <Param name="FaultOnEarlyExit" value="true" />
    </Monitor>
    <Monitor class="SaveFile">
        <Param name="Filename" value="./peachlog/fault.log" />
    </Monitor>
</Agent>

<Agent name="jacAgent">
    <Monitor class="Process">
        <Param name="Executable" value="ns" />
        <!--
        <Param name="Arguments" value="" />
        -->
        <Param name="RestartOnEachTest" value="true" />
        <Param name="FaultOnEarlyExit" value="true" />
    </Monitor>
    <Monitor class="SaveFile">
        <Param name="Filename" value="./peachlog/fault.log" />
    </Monitor>
</Agent>

<!-- define our default test case -->
```

```
        <Test name="Default">
                <Agent ref="omniAgent"/>
        <StateModel ref="stateRequest_1_0"/>
        <Publisher class="TcpClient">
            <Param name="Host" value="127.0.0.1" />
            <Param name="Port" value="2809" />
        </Publisher>
        <Strategy class="Random" />
        <!-- <Publisher class="Console" /> -->
    </Test>

    <Test name="jacorb">
        <Agent ref="jacAgent" />
        <StateModel ref="stateRequest_1_0" />
        <Publisher class="TcpClient">
            <Param name="Host" value="192.168.1.6" />
            <Param name="Port" value="2809" />
        </Publisher>
        <Strategy class="Random" />
    </Test>
</Peach>
<!-- end -->
```

Appendix F

GIOP "bind_new_context" operation Pit File

Listing F.1: GIOP "bind_new_context" operation Pit File

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns="http://peachfuzzer.com/2012/Peach"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://peachfuzzer.com/2012/Peach
    /peach/peach.xsd">

    <DataModel name="GIOPHeader_1_0">
        <String name="magic" length="4" value="GIOP" mutable="true"/>
        <Number name="major" size="8" signed="false" value="1"
            mutable="true"/>
        <Number name="minor" size="8" signed="false" value="0"
            mutable="true"/>
        <Number name="byte_order" size="8" signed="false" value="1"
            mutable="true"/>
        <Number name="message_type" size="8" signed="false" />
        <Number name="message_size" size="32" signed="false" />
    </DataModel>

    <DataModel name="GIOPRequest_1_0" ref="GIOPHeader_1_0">
        <Number name="message_type" size="8" signed="false" value="0"
            mutable="true" />
        <Number name="message_size" size="32" signed="false"
            mutable="true">
            <Relation type="size" of="Request_1_0" />
        </Number>
        <Block name="Request_1_0">
```

```
        <Number name="service_context" size="32" signed="false"
            value="0" mutable="true" />
        <Number name="request_id" size="32" signed="false"
            value="2" mutable="true" />
        <Number name="response_expected" size="8" signed="false"
            value="1" mutable="true" />
        <Number name="reserved" size="24" signed="false"
            value="0" mutable="true" />
        <Number name="object_key_length" size="32" signed="false"
            mutable="true">
            <Relation type="size" of="object_key" />
        </Number>
        <String name="object_key" value="NameService"
            nullTerminated="false" mutable="true" />
        <Number name="null1" size="8" signed="false" value="0"
            mutable="true" />
        <Number name="operation_length" size="32" signed="false"
            mutable="true">
            <Relation type="size" of="operation" />
        </Number>
        <String name="operation" value="_is_a" nullTerminated="true"
            mutable="true" />
        <Number name="operation_parameters" value="b200" valueType="hex"
            size="16" signed="false" mutable="true" />
        <Number name="requesting_principal_length" size="32"
            signed="false" value="0" mutable="true" />
        <Block name="RequestBody">
            <Number name="sequence_length" size="32" signed="false"
                value="1" mutable="true" />
            <Number name="type_id_length" size="32" signed="false"
                mutable="true">
                <Relation type="size" of="type_id" />
            </Number>
            <String name="type_id"
                value="IDL:omg.org/CosNaming/NamingContext:1.0"
                nullTerminated="false" mutable="true" />
            <String name="type_kind" value="" length="0"
                mutable="true" />
            <Number name="type_kind_length" size="8" value="0"
                mutable="true" />
        </Block>
    </Block>
</DataModel>

<DataModel name="GIOPRequest_1_0_bind" ref="GIOPRequest_1_0">
```

```
    <String name="Request_1_0.operation" value="bind_new_context"
        nullTerminated="true" mutable="true" />
    <Number name="Request_1_0.operation_parameters" value="444c3a"
        valueType="hex" size="24" signed="false" mutable="true" />
    <String name="Request_1_0.RequestBody.type_id" value="test"
        nullTerminated="true" mutable="true" />
    <String name="Request_1_0.RequestBody.type_kind" value="/Na"
        mutable="true" />
    <Number name="Request_1_0.RequestBody.type_kind_length"
        size="40" signed="false" value="1" mutable="true" />
</DataModel>

<DataModel name="GIOPReply">
    <String name="reply" value="" />
</DataModel>

    <StateModel name="stateRequest_1_0" initialState="Initial">
    <State name="Initial">
        <!--
        <Action type="output">
            <DataModel ref="GIOPRequest_1_0" />
        </Action>
        <Action type="input">
            <DataModel ref="GIOPReply" />
        </Action>
        -->
        <Action type="output">
            <DataModel ref="GIOPRequest_1_0_bind" />
        </Action>
        <Action type="input">
            <DataModel ref="GIOPReply" />
        </Action>
    </State>
</StateModel>

<Agent name="omniAgent">
    <!--
    <Monitor class="Pcap">
        <Param name="Device" value="lo" />
        <Param name="Filter" value="port 2809" />
    </Monitor>
    -->
    <Monitor class="Process">
        <Param name="Executable" value="omniNames" />
        <Param name="Arguments"
```

```
                value="-start -always -logdir ./logs
                -errlog ./logs/err.log" />
            <!--
            <Param name="RestartOnEachTest" value="true" />
            -->
            <Param name="FaultOnEarlyExit" value="true" />
        </Monitor>
        <Monitor class="CleanupFolder">
            <Param name="Folder" value="logs" />
        </Monitor>
        <Monitor class="SaveFile">
            <Param name="Filename" value="./peachlog/fault.log" />
        </Monitor>
        </Agent>

        <Test name="Default">
                <Agent ref="omniAgent"/>
        <StateModel ref="stateRequest_1_0"/>

        <Publisher class="TcpClient">
            <Param name="Host" value="127.0.0.1" />
            <Param name="Port" value="2809" />
        </Publisher>
        <Strategy class="Random" />
        <!-- <Publisher class="Console" /> -->
                <Logger class="File">
                        <Param name="Path" value="peachlog"/>
        </Logger>
        </Test>
</Peach>
<!-- end -->
```