Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing

Jared D. DeMott
Computer Science
Michigan State University
demottja@msu.edu

Richard J. Enbody
Computer Science
Michigan State University
enbody@msu.edu

William F. Punch
GARAGe
Michigan State University
punch@msu.edu

Accepted for publication at Black Hat and DEFCON 2007

Abstract

Runtime code coverage analysis is feasible and useful when application source code is not available. An evolutionary test tool receiving such statistics can use that information as fitness for pools of sessions to actively learn the interface protocol. We call this activity grey-box fuzzing. We intend to show that, when applicable, grey-box fuzzing is more effective at finding bugs than RFC compliant or capture-replay mutation black-box tools. This research is focused on building a better/new breed of fuzzer. The impact of which is the discovery of difficult to find bugs in *real* world applications which are *accessible* (not theoretical).

We have successfully combined an evolutionary approach with a debugged target to get real-time grey-box code coverage (CC) fitness data. We build upon existing test tool General Purpose Fuzzer (GPF) [8], and existing reverse engineering and debugging framework PaiMei [10] to accomplish this. We call our new tool the *Evolutionary Fuzzing System* (EFS), which is the initial realization of my PhD thesis.

We have shown that it is possible for our system to learn the targets language (protocol) as target communication sessions become more fit over time. We have also shown that this technique works to find bugs in a real world application. Initial results are promising though further testing is still underway.

This paper will explain EFS, describing its unique features, and present preliminary results for one test case. We will also discuss ongoing research efforts. First we begin with some background and related works.

Previous Evolutionary Testing Work

"Evolutionary Testing uses evolutionary algorithms to search for software test data. For white-box testing criteria, each uncovered structure-for example a program statement or branch-is taken as the individual target of a test data search. With certain types of programs, however, the approach degenerates into a random search, due to a lack of guidance to the required test data. Often this is because the fitness function does not take into account data dependencies within the program under test, and the fact that certain

program statements need to have been executed prior to the target structure in order for it to be feasible. For instance, the outcome of a target branching condition may be dependent on a variable having a special value that is only set in a special circumstance-for example a special flag or enumeration value denoting an unusual condition; a unique return value from a function call indicating that an error has occurred, or a counter variable only incremented under certain conditions. Without specific knowledge of such dependencies, the fitness landscape may contain coarse, flat, or even deceptive areas, causing the evolutionary search to stagnate and fail. The problem of flag variables in particular has received much interest from researchers (Baresel et aL, 2004; Baresel and Sthamer, 2003; Bottaci, 2002; Harman et aL, 2002), but there has been little attention with regards to the broader problem as described. [1]"

The above quote is from a McMinn paper that is pushing forward the field of traditional evolutionary testing. However, in this paper we propose a method for performing evolutionary testing (ET) that does not require source code. This is useful for third-party testing, verification, and security audits when the source code of the test target will not be provided. Our approach is to track the portions of code executed ("hits") during runtime via a debugger. Previous static analysis of the compile code, allows the debugger to set break points on functions (funcs) or basic blocks (BBs). We partially overcome the traditional problems of evolutionary testing by the use of a seed file, which gives the evolutionary algorithm hints about the nature of the protocol to learn. Our approach works differently from traditional ET in two important ways:

1. We use a grey-box style of testing that allows us to proceed without source code
2. We search for sequences of test data, known as sessions, which fully define the documented and undocumented features of the interface under test (protocol discovery). This is very similar to finding test data to cover every source code branch via ET. However, the administration, of discovered test data is happening during the search. Thus, test results, are discovered as our algorithm runs. Robustness issues are recorded in the form of crash files and Mysql data, and can be further explored for exploitable conditions while the algorithm continues to run.

Introduction

Fuzzing is simply another term for interface robustness testing. Robustness testing often indicates security testing of user accessible interfaces, often called the *attack surface*. This is not security testing in the sense that a penetration test is being performed. We're testing if user supplied input validation errors exist (think buffer overflows and the like). Solid security in said target is not possible if such validation errors are found. Fuzzing does not replace formal engineering practices, solid quality assurance, or a full code audit and penetration test. EFS focuses on testing the robustness of a given attack surface in the face of unexpected input.

Solid work has been done in the field of software testing. Much work has also been done in the field of white-box evolutionary testing [1] [5]. Our work is unique in that no other

grey-box fuzzer using evolutionary computing to generate test cases is known at this time. White-box indicates access to source code. Black-box indicates the ability to supply data to a running program, but no source code. In Grey-box while no access to source code is directly granted, it is possible to monitor the running executable in as much detail as a debugger and/or static binary analysis will permit.

Current Fuzzers

Current fuzzer development has two main branches: full and mutation. A full fuzzer uses a protocol specific (think RFC) to the target program and works only for that protocol. For example, one might develop a fuzzer specific to SMTP. A mutation fuzzer (sometimes called capture/replay) starts with some known good data, changes it somehow, and than repeatedly delivers mutations of that data to the target. Many fuzzers will also monitor the state of the application during fuzzing and report access violations. Both types of fuzzers have value. The full fuzzer will typically get better code coverage (and thus find more bugs), but the mutation fuzzer is quicker to develop and could uncover bugs the full might not. For example, the mutation fuzzer might create an SMTP conversation with a target using commands not listed in the SMTP RFC.

Open Source vs. Commercial

The debate within fuzzing lists [14], quality assurance groups, security conferences, and testers of all kinds rages about who owns the state of the art here. Commercial companies claim they do. Most vender neutral testers say open source solutions are superior. There has been no real study to date. This would make an excellent study. If time permits, and vendors cooperate, we would like to do such a study.

EFS Overview

We propose a new fuzzer which we call the Evolving Fuzzer System or EFS as shown in Figure 1. We'd like to receive the benefits of both fuzzer types: good code coverage and short development time per application.

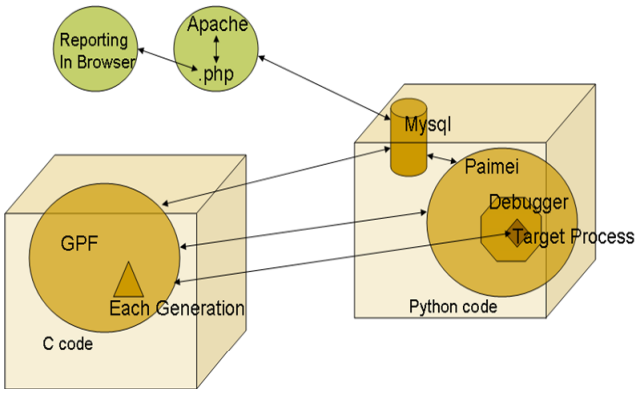

Figure 1: The Evolving Fuzzer System (EFS)

EFS will learn the target protocol by evolving *sessions*: a sequence of input and output that makes up a conversation with the target. To keep track of how well we are doing we

use code coverage as a session metric (fitness). Sessions with greater fitness breed to produce new sessions. Over time, each generation will cover more and more of the code in the target. In particular, since EFS covers code that can be externally exercised, it covers code on the networked attack surface. EFS could be adapted to fuzz almost any type of interface (attack surface). To aid in the discovery of the language of the target, a seed file is one of the parameters given to the GPF portion of EFS (see Figure 8). The seed file contains binary data or ASCII strings that we expect to see in this class of protocol. For example, if we're testing SMTP some strings we'd expect to find in the seed file would be: "helo", "mail to: ", "mail from: ", "data", "\r\n.r\n", etc. EFS could find the strings required to speak the SMTP language, but for performance, initialing some sessions with known requirements (such as a valid username and password, etc.) will be beneficial.

EFS uses fuzzing heuristics in mutation to keep the fuzzer from learning the protocol completely correct. Fuzzing heuristics include things like bit-flipping, long string insertion, format string creation, etc. Probably even more important is the implicit fuzzing that a GA performs. Many permutations of valid command orderings will be tried and retried with varying data. The key to fuzzing is the successful delivery, and subsequent consumption by the target, of *semi-valid* sessions of data. Sessions that are entirely correct will find no bugs. Sessions that are entirely bogus will be rejected by the target. Testers might call this activity "good test case development".

While the evolutionary tool is learning the unfamiliar network protocol, it may crash the code. That is, as we go through the many iterations of trying to learn each layer of a given protocol we will be implicitly fuzzing. If crashes occur, we make note of them and continue trying to learn the protocol. Those crashes indicate places of interest in the target code for fixing or exploiting depending on which hat is on. The probability of finding bugs, time to convergence, and *total diversity* are still under research at this time.

A possible interesting side effect of automatic protocol discovery is the iteration paths through a give protocol. Consider for example the recent VNC bug. The option to use no authentication was a valid server setting, but should never have been possible to exercise from the client side unless specifically set on the server side. However, this bug allowed a VNC client to choose no authentication even when the server was configure to force client authentication. This allowed a VNC client to control any VNC server (of a specific release version) without valid credentials. This notion indicates that it might be possible to use EFS results, even if no robustness issues are discovered, to uncover possible security or unintended functionality errors. Data path analysis of the matured sessions would be required at the end of a run.

Total diversity is perceived to be an important metric leading to maximum bug discovery capability. Diversity indicates the percentage of code coverage on the current attack surface. If EFS converges to one best session, and than all other sessions begin to look like that (which is common in genetic algorithms), this will be the only path through code that is thoroughly tested. Thus, it's important to measure diversity while testing. As a method to test such capabilities a benchmarking system is in development. Initial results

are interesting and indicate that the use of multiple pools to store sessions is helpful in maintaining a slightly higher level of diversity. However, maximum diversity (total attack surface coverage) was not possible with pools. We intend to develop a newer *niching* or *speciation* technique, which will measure the individuality of each session. Those that are significantly different from the best session, regardless of session fitness, will be kept. (I.e., they will be exempt from the crossover process). In this case, the simple fitness function we use now (hit basic blocks or functions) would be a little more complex. Again, it would than consider session uniqueness [15].

GPF + PaiMei + Jpgraph Reporting + Countless Hours of Implementation = EFS:
We choose to build upon GPF because the primary author of this paper is also the author of that fuzzer, and consequently controls access to the source code. GPF was designed to fuzz arbitrary protocols given a capture of real network traffic. In this case, no network sniff is required, as EFS will learn the protocol dynamically.

PaiMei was chosen because if it's ability to "stalk" a process. The process of stalking involves:

- Pre-analyzing an executable to find functions and basic blocks
- Attach to that executable as it runs and set breakpoints.
- Checking off those breakpoints as they are *hit*.

GPF and PaiMei had to be substantially modified to allow the realization of EFS. PHP code, using the Jpgraph library, was written to access the database to build and report graphical results.

EFS Data Structures

A *session* is one full transaction with the target. A session is made up of *legs* (reads or writes). Each leg is made up of *tokens*. A token is a piece of data. Each token has a type (ASCII, BINARY, LEN, etc.) and some data ("jared", \xfe340078, etc.). Sessions are organized into pools of sessions. See Figure 2. This organization is for data management, but we also maintain a pool fitness, the sum of the unique function hits found by all sessions. Thus, we maintain two levels of fitness for EFS: session fitness and pool fitness. We maintain pool fitness because it is reasonable that a group of lower fit sessions, when taken together, could be better at finding bugs than any single, high-fit session. In genetic algorithm verbiage [7], each chromosome represents a communication session.

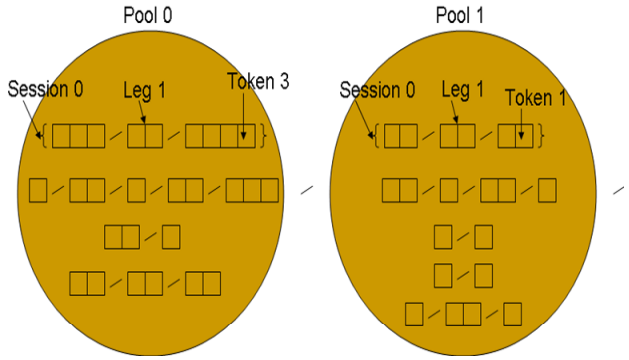

Figure 2: Data Structures in EFS

EFS Initialization

Initially, p pools are filled with at most s-max sessions each of which has at most l-max legs each of which has at most t-max tokens. The type and data for each token are drawn 35% of the time from a seed file or 65% of the time randomly generated. Again, a seed file should be created for each protocol under test. If little is known about the protocol a generic file could be used, but pulling strings from a binary via reverse engineering, or sniffing actual communications is typically possible. Using no seed file is also a valid option.

For each generation, every session is sent to the target and a *fitness* is generated. The fitness is coverage which we measure as the number of functions or basic blocks hit in the target. At the end of each generation, evolutionary operators are applied. The rate (every x generations) at which session mutation, pool crossover, and pool mutation occurs is configurable. Session crossover occurs every generation.

Session Crossover

Having evaluated code-coverage/fitness for each session, we use the following algorithm for crossover (see Figure 3):

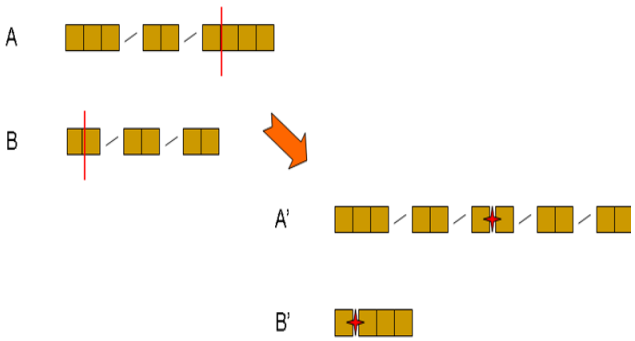

Figure 3: Session Crossover

1. Order the sessions by fitness, with the most fit being first.
2. The first session is copied to the next generation untouched. Thus we do use elitism.

3. Randomly pick two parents, A and B, and perform single point crossover, creating children A' and B'. Much like over-selection in genetic programming, 70% of the time we use only the top half of the sorted list to pick parents from. 30% of the time we chose from the entire pool.
4. Copy all of the A Legs into A' up until the leg that contains the cross point. Create a new leg in A'. Copy all tokens from current A leg into the new A' leg, up until the cross point. In session B advance to the leg that contains the cross point. In that leg advance to the token after the cross point. From there, copy the remaining tokens into the current A' leg. Copy all the remaining legs from B into A'.
5. If we have enough sessions stop. Else,
6. Create B' from (B x A)
7. Start in B. Copy all of the B Legs into B' up until the leg that contains the cross point. Create a new leg in B'. Copy all tokens from that B leg into the new B' leg, up until the cross point. In session A advance to the leg that contains the cross point. In that leg advance to the token after the cross point. From there, copy the remaining tokens into the current B' leg. Copy all the remaining legs from A into B'.
8. Repeat until our total number of sessions (1st + new children) equals the number we started with.

Session Mutation

Since we are using elitism, the elite session is not modified. Otherwise, every session is potentially mutated with probability p. The algorithm as follows (example in Figure 4):

Figure 4: Session Mutation

1. For each session we randomly choose a leg to do a data mutation on. We then randomly choose another leg to do a type mutation on.
2. A Data mutation modifies the data in one random token in the chosen leg. Fuzzing heuristics are applied, but a few rules are in place to keep the tokens from growing to large.
3. If the token is too large or invalid, we truncate or reinitialize.
4. The heuristics file also contains the rules detailing how each token is mutated. For example a token that contains strings (ASCII, STRING, ASCII_CMD, etc) is more likely to be mutated by the insertion of a large or format string. Also, as part of the information we carry on each token we will know if each token

contains specific ASCII traits such as numbers, brackets, quotes, etc. We may mutate those as well. Tokens of type (BINARY, LEN, etc.) are more likely to have bits flipped, hex values changed, etc.

5. The type mutation has a chance to modify both the type of the leg and the type of one token in that leg. Leg->type = _rand(2) could reinitialize the legs type. (That will pick either a 0 or a 1. 0 indicates READ and 1 indicates WRITE.) tok->type = _rand(14) could reinitialize the tokens type. There are 0-13 valid types. For example, STRING is type 0. (structs.h contains all the definitions and structure types.)

Pool Crossover

Pool crossover is very similar to session crossover, but the fitness is measured differently. Pool fitness is measured as the sum of the code uniquely covered by the sessions within. That is, count all the unique functions or basic blocks hit by all sessions in the pool. This provides a different (typically better) measure than say the coverage by the best session in the pool. See Figure 5.

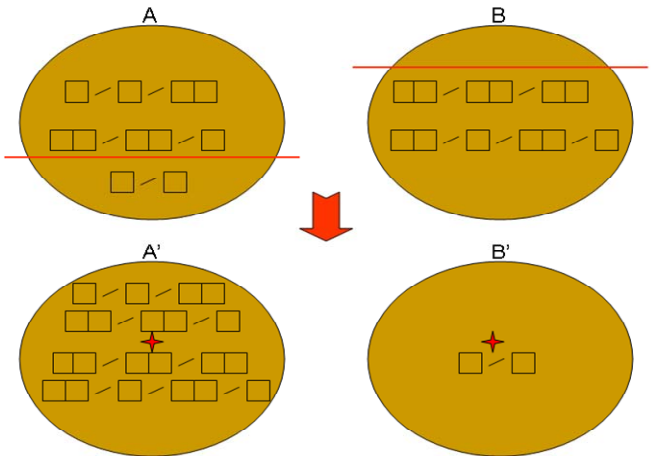

Figure 5: Pool Crossover

The algorithm is:

1. Order the pools by fitness, with the most fit being first. Again, pool fitness is the sum of all the sessions' fitness.
2. The first pool is copied to the next generation untouched. Thus elitism is also operating at the pool level
3. Randomly pick two parents and perform single point crossover. The crossover point in a pool is the location that separates one set of sessions from another. 70% of the time we use only the top half of the sorted list to pick parents from. 30% of the time we chose from the entire list of pools.
4. Create A' from (A x B):
5. Start in A. Copy all of the sessions from A into A' up until the cross point. In pool B, advance to the session after the cross point. From there, copy the remaining sessions into A'.
6. If we have enough pools stop. Else,
7. Create B' from (B x A)

8. Start in B. Copy all of the sessions from B into B' up until the cross point. In pool A, advance to the session after the cross point. From there, copy the remaining sessions into B'.
9. Repeat until our total number of pools (1st + new children) equals the number we started with.

Pool Mutation

As with session mutation, pool mutation does not modify the elite pool. The algorithm is (example in Figure 6):

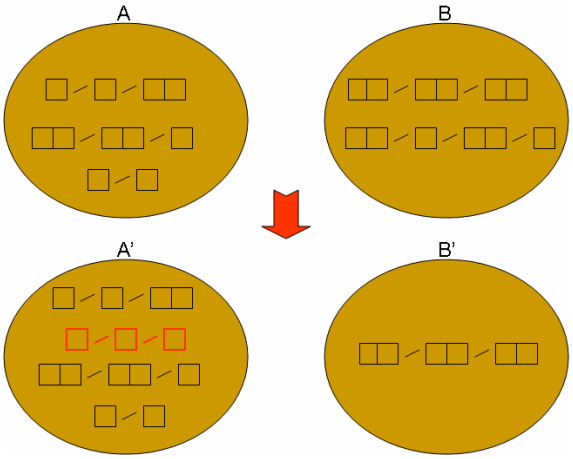

Figure 6: Pool Mutation

1. 50% of time we add a session according to the new session initialization rules.
2. 50% of the time we delete a session.
3. If the sessions/pool are fixed, we do both.
4. In all cases, we don't disturb the first session.

Running EFS

From a high level, the protocol between EFS-GPF and EFS-PaiMei is as follows:

GPF initialization/setup data → PaiMei
Ready ← PaiMei
<GPF carries out communication session with target>
GPF {OK|ERR} → PaiMei
<PaiMei stores all of the hit and crash (if any) information to the database>

When all of the sessions for a given generation have been played GPF contacts the database, calculates a fitness for each session (counts hits) and for each pool (distinct hits for all sessions within a pool), and breeds sessions and pools as indicated by the configuration options (See the description of Figure 8).

Figures 7 and 8 show the EFS-GPF and EFS-PaiMei portions of EFS in action. For the GUI portion we see:

1. Two methods to choose an executable to stalk:

    a. The first is from a list of process identifications (PIDs). Click the "Refresh Process List" to show running processes. Click the process you wish to stalk.
    b. The second is by specifying the path to the executable with arguments. An example would be: *"c:\textserver.exe" med*
2. We can choose to stalk *functions* (funcs) or *basic blocks* (BBs).
3. The time to wait for each target process load defaults to 6 seconds, but could be much less (1 second) in many cases.
4. Hits can be stored to the *GPF* or *PaiMei* sub-databases that are in the Mysql database. PaiMei should be used for tests or creating filter tags, while GPF should be used for all EFS runs.
5. After each session, or stalk, we can do *nothing*, *detach* from the process (and reattach for the next stalk), or *terminate* the process. The same options are available if the process crashes.
6. Use the *PIDA Modules* box for loading the .pida files. These are derived from executables or dynamically linked libraries (.DLLs), and are used to set the breakpoints which enable the process stalking to occur. One executable needs to be specified and as many .DLLs as desired. (Note: Sometimes processes will include files called .api, .apl, etc which are really .DLLs and can be used here as well.)
7. There is a dialog box under *Connections* to connect to the Mysql database. Proper installation and setup of EFS-PaiMei (database, etc.) is included in a document in the EFS source tree.
8. The *Data Sources* box is the place to view target lists and to create filter tags. This is done to speed up EFS, by weeding out hits that are common to every session. The process to create a filter tag is:
    a. Define a filter tag. (We called ours "ApplictionName_startup_conn_junk_disconn_shutdown")
    b. Stalk with that tag and record to the PaiMei database
    c. Start the target application
    d. Using netcat, connect to the target application
    e. Send a few random characters
    f. Disconnect
    g. Shutdown the target application
9. There is another dialog box that defines the GPF connection to EFS-PaiMei called *Fuzzer Connect*.
    a. The default port is 31338 (if you don't get why that number, ask a hax0r).
    b. The general wait time describes how long each session has to complete before EFS will move on to the next session. This is needed to coordinate the hit dumping to mysql after each session. The default is .8 but for lean applications running around .2 should be fine. For larger applications more time will be required for each session. Tuning this number is the key to the speed that EFS will run at. (For example: .4*100000=11hrs, .8*100000=22hrs, 1.6*100000=44hrs, etc)

c. The “dump directory” defines a place for EFS to dump crash information should a robustness issue be found. We typically create a directory of the structure “..\EFS_crash_data\application_name*number*”.
d. The *number* should coordinate to the GPF_ID for clarity and organization.

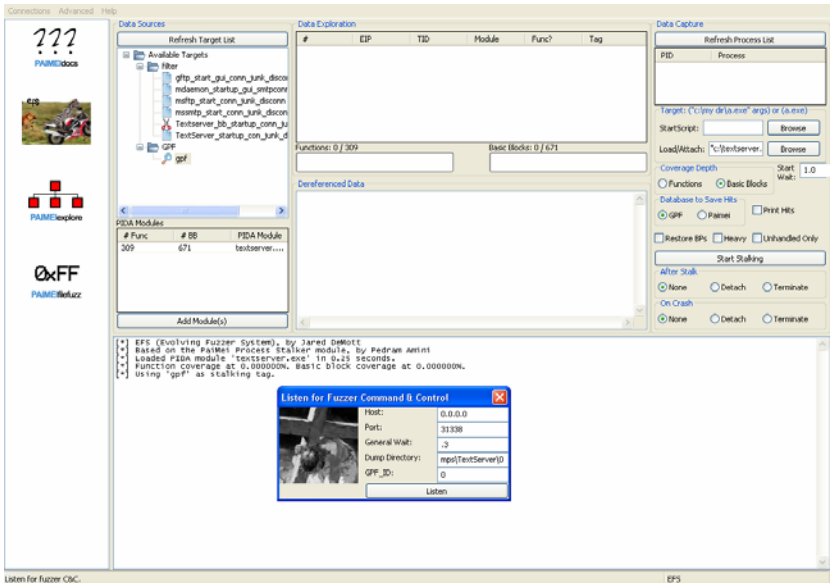

Figure 7: The GUI portion of EFS

For the GPF (command line) portion of EFS we have 32 options:

1. –E indicates GPF is in the evolving mode. GPF has other general purpose fuzzing modes which will not be detailed here.
2. IP of Mysql db
3. Username of Mysql db
4. password for Mysql db
5. GPF_ID
6. Starting generation. If a number other than zero is specified, a run is picked up where it left off. This is helpful if EFS where to crash, hang, or quit.
7. IP of GUI EFS
8. Port of GUI EFS
9. Stalk type. Functions or basic blocks.
10. Play mode. *Client* indicates we connect to the target and *server* is the opposite.
11. IP of target. (Also IP of proxy in proxy mode.)
12. Port of target. (Also port of proxy in proxy mode.)
13. Source port to use. ‘?’ lets the OS choose.
14. Protocol. TCP or UDP
15. Delay time in milliseconds between each leg of a session.
16. Number of .01 seconds slots to wait while attempting to read data.
17. Output verbosity. Low, med, or high.
18. Output mode. Hex, ASCII, or auto.
19. Number of pools.
20. Number of sessions/pool.

21. Is the number *fixed* or a *max*? Fixed indicates it must be that number while max allows any number under that to valid as well.
22. Legs/session
23. Fixed or max
24. Tokens/leg
25. Fixed or max
26. Total generations to run
27. Generation at which to perform session mutation
28. Generation at which to perform pool crossover
29. Generation at which to perform pool mutation
30. User definable function on outgoing sessions. *None* indicates there isn't one.
31. Seed file name.
32. Proxy mode. *Yes* or *no*. A proxy can be developed to all EFS to run against none network protocols such as internal RPC API calls, etc.
33. (UPDATE: A 33rd was just added to control diversity.)

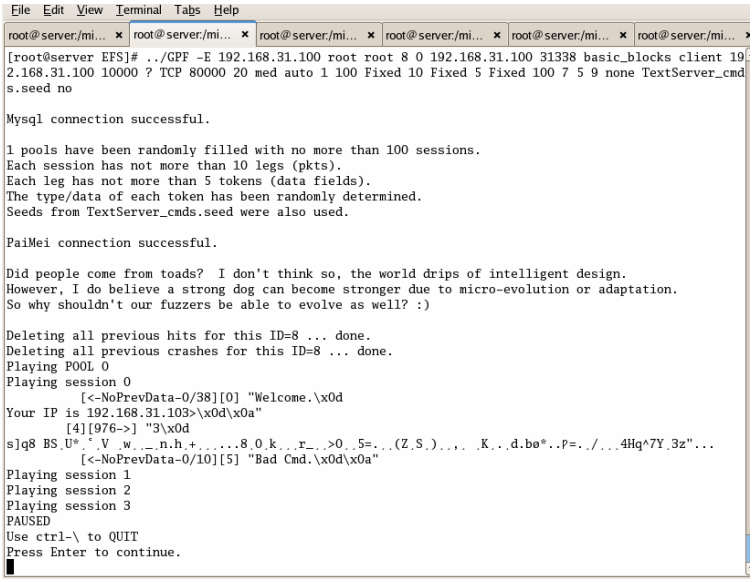

Figure 8: The GPF UNIX command line portion of EFS

Benchmarking

The work in this section has become intense enough to warrant a whole new paper. See *Benchmarking Grey-box Robustness Testing Tools with an Analysis of the Evolutionary Fuzzing System (EFS)* [15]. The topics in that paper include:

- Attack surface example
- Functions vs. basic blocks.
- Learning a binary protocol
- Pools vs. niching
    - EFS Fitness function updates to achieve greater diversity

Test Case – Golden FTP server

The first test target was the Golden FTP server (GFTP) [9]. It is a public domain ftp server GUI application for Windows that has been available since 2004. Analysis shows approximately 5100 functions in GFTP, of which about 1500 are concerned with the GUI/startup/shutdown/config file read/etc, leaving potentially 3500 functions available. However, the typical attack surface of a program is considerably smaller, often around 10%. We show more evidence of this in the benchmarking research.

Three sets of experiments were run. Each experiment was run 3 times on two separate machines (6 total runs/experiment). The reason for two machines was two fold: time savings, as each complete run can take about 6hrs/100generations, and to be sure configurations issues were not present on any one machine. Experiment 1 is 1 pool of 100 sessions. Experiment 2, 4 pools each with 25 sessions. Experiment 3, 10 pools each with 10 sessions. All other parameters remain the same: target was Golden Ftp Server v1.92, 10 legs/session, 10 tokens/leg, 100 total generations, session mutation every 7 generations, for multiple pool runs—pool crossover every 5 generations, and pool mutation every 9 generations. For these experiments we used function hits as the code coverage metric. The session, leg, and token sizes are fixed values.

Results

Figure 9 shows the average fitness for both pool and session runs, averaged over all the runs for each group. Figure 10 shows the best fitness for both pool and session, selected from the "best" run (that is, the best session of all the runs in the group, and the best pool of all the runs in the group). The first thing that Figure 9 shows us is that pools are more effective at covering code than any single session. Even the worst pool (1-pool) covers more code than the best session. Roughly speaking, the best pool covers around twice as much as the best session. The second observation that Figure 9 shows us is that multiple, interacting pools are more effective than a single large pool. Note that this is not just a conclusion about island-parallel evolutionary computation [11], since the interaction between pools is more frequent and of a very different nature than the occasional exchange of a small number of individuals as found in island parallelism. The pool interaction is more in line with a second-order evolutionary process, since we are evolving not only at the session level, but also at the pool level. While pool-1 starts out with better coverage, it converges to less and less coverage. Both 4-pool and 10-pool start out with less coverage, but have a positive fitness trajectory on average, and 4-pool nearly equals the original 1-pool performance by around generation 180 and appears to still be progressing.

Figure 10 shows that, selecting for the best pool/session from all the runs (not the averages as in Figure 5), 4-pool does slightly outperform other approaches. That is, the best 4-pool run outperformed any other best pool, and greatly outperformed any best session.

The information provided by Figures 11, 12, and 13 shows the following: First, they show the total number of crashes that occurred across all runs for 1-pool, 4-pool, and 10-pool. The numbers around the outside of the pie chart are the actual number of crashes

that occurred for that piece, while the size of each pie chart piece indicates that crash's relative frequency with respect to all crashes encountered. Furthermore, the colors of each piece reflect the addresses in gftp.exe where the crashes occurred. Remember that the only measure of fitness that EFS uses is the amount of code covered, not the crashes. However, these crash numbers provide a kind of history of the breadth of search each experiment has developed. For example, all 3 experiments crashed predominantly at address 0x7C80CF60. However, 10-pool found a number of addresses that neither of the others did, for example the other 0x7C addresses.

GFTP is an interesting (and obviously buggy) application. In creates a new thread for each connection, and even if that thread crashes can keep processing the current session in a new thread. This allows for multiple crashes/session, something that was not originally considered. This accounts for the thousands of crashes observed. Also, keep in mind these tests are done in a lab environment, not on productions systems. Nothing was affected by our crashes, or could have caused them. These tests were done in January 2007, and no ongoing effort against GFTP is in place to note rather or not these bugs have been patched. Also, no time was spent attempting to develop exploits from the recorded crash data. It is the authors' opinion that such exploits could be developed but we would rather focus on continued development and testing of EFS.

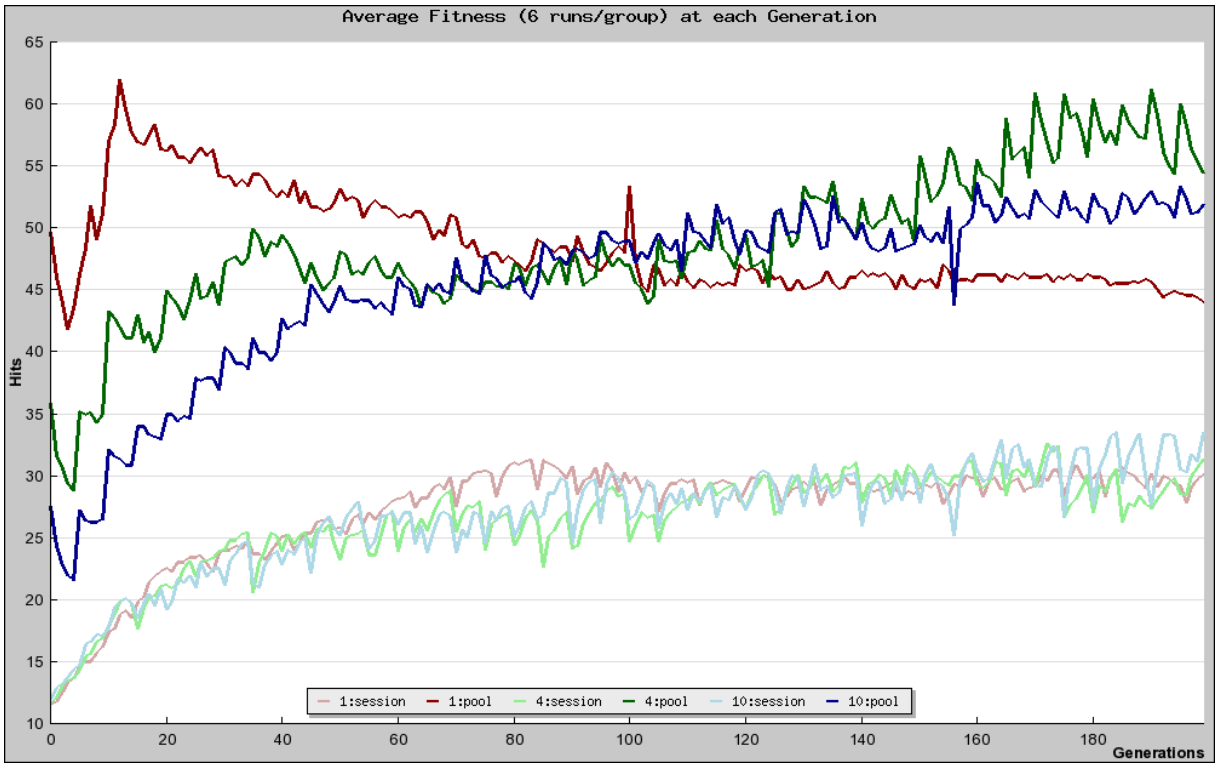

Figure 9: Average Fitness of pool and session over 6 runs

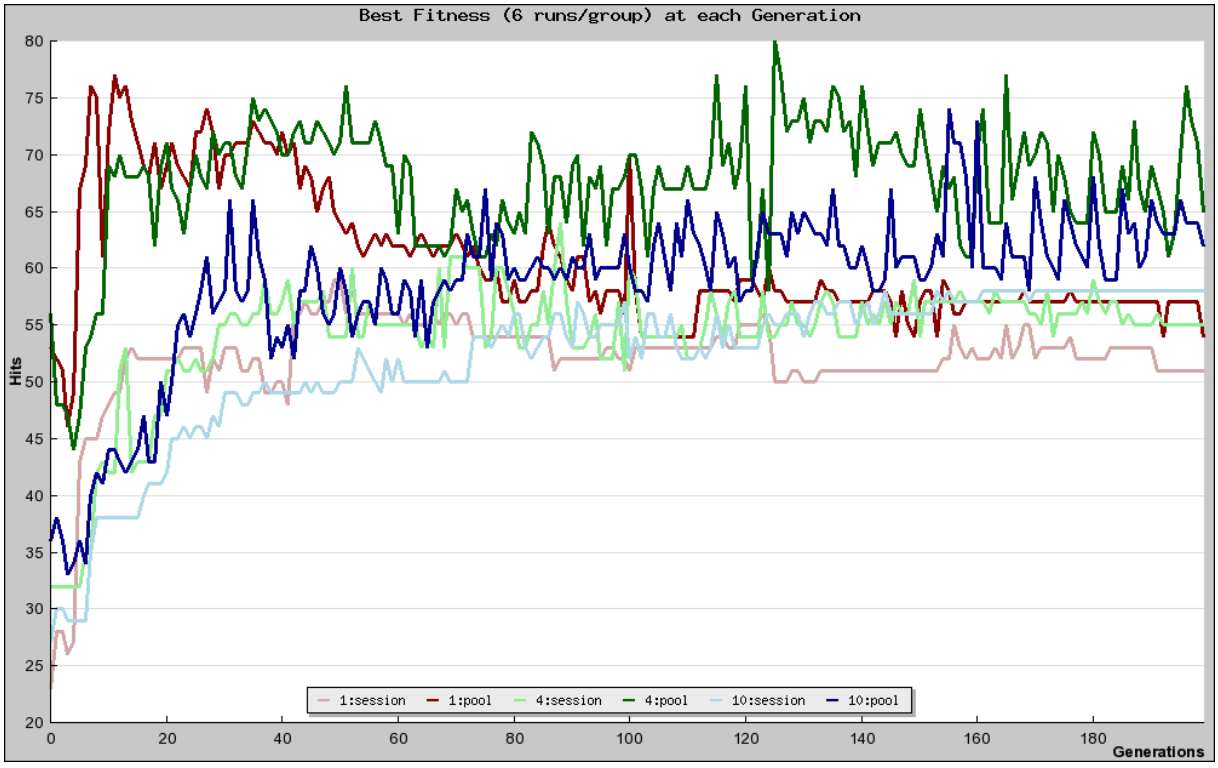

Figure 10: Best of Pool and Session over 6 Runs

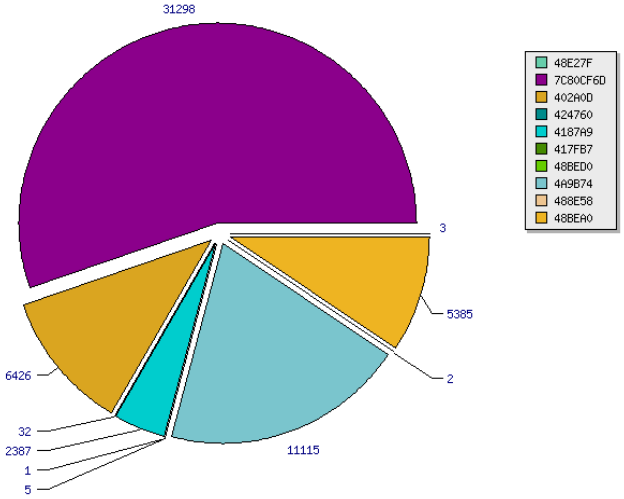

Figure 11: 1-pool Crash Total (all runs)

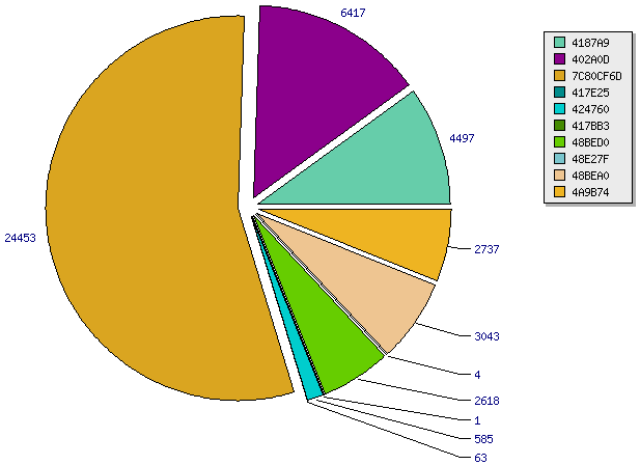

Figure 12: 4-pool Crash Total (all runs)

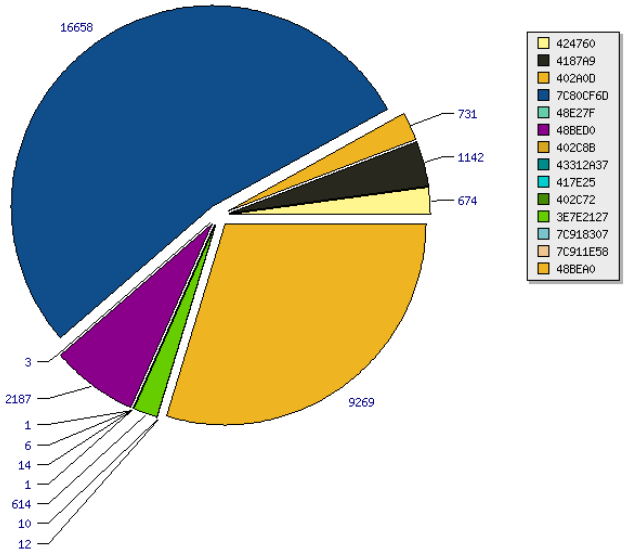

Figure 13: 10-pool Crash Total (all runs)

Conclusions and Future Work

We have shown that EFS was able to learn a protocol and find bugs in real software using a grey-box evolutionary robustness testing technique. Continuing research:

- What is the probability to find various bug types as this is the final goal of this research
  - How does its performance compare with existing fuzzing technologies?
  - What bugs can be found and in what software?
- Could this type of learning be important to other fields?
- Is it possible to cover the *entire* attack surface with our approach? How would one know, since we don't have the source code?
  - Pools don't seem to have completely covered the target interface, is their a *niching* or *speciation* approach we can design ?
- Testing of clear text protocols was done, but is it also possible to learn more complex binary protocols?

References

[1] McMinn, P. “Search-based Software Test Data Generation: A Survey”. Software Testing, Verification & Reliability, Vol 14, Num 2, pp 105-156, 2004

[2] Roper, M. “Computer aided software testing using genetic algorithms”, in 10th International Software Quality Week, San Francisco, 1997

[3] Watkins, A., “The automatic generation of test data using genetic algorithms”, in Proceedings of the Fourth Software Quality Conference, pp 300-309, 1995

[4] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities", Communications of the ACM 33, 12 (December 1990). See also http://www.cs.wisc.edu/~bart/fuzz/

[5] P. McMinn and M. Holcombe, “Evolutionary Testing Using an Extended Chaining Approach”, ACM Evolutionary Computation, Pgs 41-64, Volume 14, Issue 1 (March 2006)

[6] Stefan Wappler, Joachim Wegener: Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. GECCO 2006: 1925-1932

[7] Goldberg, David E. Genetic Algorithms in Search, Optimization and Machine Learning Addison-Wesley Pub. Co. 1989. ISBN: 0201157675

[8] http://www.appliedsec.com/resources.html

[9] http://www.goldenftpserver.com/

[10] Pedram Amini, PaiMei Reverse Engineering Framework, http://pedram.redhive.com/PaiMei/

[11] Cantu-Paz, E. “Efficient and Accurate Parallel Genetic Algorithms”, Kluwer Academic Publishers, 2000

[12] Pargas, Harrold, & Peck. “Test-Data Generation Using Genetic Algorithms”, Journal of Software Testing, Verification and Reliability, 1999.

[13] Wegener, Sthamer, & Baresel. “Application Fields for Evolutionary Testing”, EuroSTAR, 2001.

[14] A mailing list dedicated to the discussion of fuzzing. fuzzing@whitestar.linuxbox.org

[15] J. DeMott, “Benchmarking Grey-box Robustness Testing Tools with an Analysis of the Evolutionary Fuzzing System (EFS)”, continuing PhD research