

# Benchmarking Grey-box Robustness Testing Tools with an Analysis of the Evolutionary Fuzzing System (EFS)

Jared DeMott  
[demottja@msu.edu](mailto:demottja@msu.edu)

## **Background and Introduction**

The goal of this project is to create two network programs that can be used to measure the effectiveness of automated grey-box robustness testing tools. One program (*TextServer.exe*) will operate as a text or string based protocol, mimicking the likes of FTP, SMTP, NMAP, etc. The other program (*BinaryServer.exe*) will run using a binary protocol, testing how a particular grey-box tool might perform against the many structured protocols such as DNS, RPC, IKE, SIP, etc. We assume it'll be more difficult to dynamically learn a binary protocol.

These two categories were chosen as benchmark metrics because they represent the two most common network application interface protocols. From a high level every network protocols falls into one of these two categories.

Both protocols will include multiple paths that could be followed. Both protocols will also have code errors in predetermined positions. Our basic premise is that tools and/or tool settings that best cover code *and* finds the errors, wins. The catch here is that tools have to be grey-box. In other words, it's not fair to build a test tool that knows the protocol and goes directly to the errors. Why test if we know where the errors are? And while writing fuzzers that are RFC compliant is a very feasible, useable, and effective way to test, we choose not to tackle the problem in that manner. We operate gray-box style because it's often desirable to test proprietary protocols with no access to source code. Also, because building or buying a tool for each known protocol is arguably more expensive.

## **The Evolutionary Fuzzing System (EFS)**

In particular, these two programs will help us better study and understand how the Evolutionary Fuzzing System (EFS) works. EFS uses *sessions* that are played to a *target* (subject under test). The target is monitored by a debugger that exports code coverage (CC) metrics to a database after each session. Sessions can be organized into multiple *pools*, which allow for non-standard co-evolution. There are many other parameters, including *legs/session*, *tokens/leg*, generations between evolutionary events, etc., that can be tuned. As of now, we have little evidence to indicate which settings might be optimal. (We do have some empirical evidence, showing the benefit of multiple pools, from a previous study of the Golden FTP server [1].) One reason for this is the complexity and non-relevant code in open source or compiled commercial daemons that are potential test subjects. Thus, it seemed wise to study code we control and understand.

The primary focus of EFS is to create a fuzzer that is better at finding bugs than current fuzzers. We believed that session diversity will help accomplish this, because more code will be regularly covered if the sessions maintain diversity over time. We know that code coverage does not guarantee that the right paths will be covered, in the right order, with the right data, but if code hasn't been covered bugs haven't been searched for. Operating with that knowledge it seems wise to cover code over and over again with varying data and in varying orders. We believe the basic "slamming around" nature of genetic algorithms will assist with this notion and help facilitate bug discovery. Fuzzing heuristics are present in the data mutations that take place within sessions. We rely on that for the formation of semi-valid (partially malformed, but mostly correct) data. Another benefit of CC, received by EFS, is the real-time knowledge that sessions are being consumed by the target. If we notice a significant drop in fitness, we would readily know we've run into something such as a timing problem with the target. Or, perhaps the target has black listed the fuzzer IP address because of too many concurrent connections.

One assumption we are operating under is that the more possible paths through code, the more effective pools will be. This does not imply that more paths should equal more pools, it simply assumes that more than one pool will be increasingly helpful as the number of paths increases. To test this assumption *TextServer* can operate in three modes: *low* (1 path), *med* (9 paths), and *high* (19 paths).

The assumption that coincides with the above is that the settings that allow us to maximize diversity will be better at finding a greater number (diversity) of bugs. If it is determined that pools do not help us maintain a proper amount of diversity, it will likely be best to run with one large pool, and add *niching* (might also be called *speciation*) functionality to the fitness evaluation. That is, sessions that have hits  $x\%$  different than the best session are directly carried over regardless of fitness, or perhaps given some fitness boost.

One CC related question we've had for sometime was rather or not evolving on functions hit or basic blocks hit would be better. A function is a normal function call and a basic block is a sequence of assembly instructions before a branch. The problem is that there are many more basic blocks than functions, thus slowing down the testing process substantially. A server can be slowed so much that normal operation is affected. However, that will not be the case in our test servers due to the uncomplicated nature of the code. We should be able to test if having a finer fitness function is helpful or not. Some would automatically assume it would be. However, due to the "higher level" of evolution here, it may not be. Higher level indicates that the CC oriented fitness function is not able to fully drill down into the fitness landscape. There has been some good work in white-box testing to refine this process [2]. But since we are using gray-box, those techniques are not applicable. We have to rely on the *seedfile* to overcome flat areas in the fitness landscape. We had once considered moving to a MSR tracing technique that would allow us to record all instructions hit, with very little slow down. This would also allow us to do away with the sometimes painful PIDA (pre-analysis to find functions (funcs) and basic blocks (BBs)) step. However this technique would not be perfect either. A few things to consider:

1. MSR doesn't work in virtual machines such as VMWare
2. All instructions are traced so EFS would have to manually filter hits outside the scope of the target DLL(s). (I.e. the many jumps to kernel and library DLLs.)
3. Pre-analysis is still required to determine how many total funcs/BBs there are if the percent of CC is desired.
4. It's not possible to "stop" watching a Func/BB. The PIDA option allows the EFS operator to select to *not* restore break points after they have been hit once. So while the PIDA option is slower to create and start up, it could be faster over time if only a small number of breakpoints are set in the target.

## ***Text Protocol***

Below is a description of the basic protocol:

```

← "Welcome.\r\n Your IP is 192.168.31.103"
"cmd x\r\n" →
← "Cmd x ready. Proceed.\r\n"
"y\r\n" →
← "Sub Cmd y ok.\r\n"
"calculate\r\n" →
← "= x + y\r\n"

```

Figure 1: The *TextServer* Protocol

Replace *x* and *y* with 0-9. On *high*, this will create 19 valid paths through code, and 4 total error conditions, which have the form of: "Bad Cmd.\r\n", "Bad Sub Cmd number.\r\n", "Bad Cmd Completion\r\n", and "Too Many Bad Cmds.\r\n"win

## **Investigating Two Basic Assumptions (why switch fonts?)**

Figures 2 and 3 are intended to illustrate a couple basic assumptions: If there are not enough functions to drive the evolutionary algorithm it'll be difficult to learn and if there is only one path, no pools or other diversity measures are required.

The first four runs test the *TextServer* on *low*, using funcs and BB, 1 pool and 4 pools. Figures 2 and 3 use funcs, and could have found a best session of 6 *hits*. An example of how the "hit" numbers come about is detailed in the next section. After 100 generations neither run found the best session. Figure 2 shows that total diversity is never higher than the single best session. Since there is only one path, we know that diversity is not possible, for the *low* setting.

In Figures 4 and 5 we see the effects of adding basic blocks. Figure 4 is no surprise; we see the total diversity converging toward the single best session. This is due to using a single pool. This issue will be addressed further.

For Figures 4 and 5, a correct session on *low* ({cmd 0 ->}, {<- ok}, {0 ->}, {<-ok} {calculate ->}, {<-answer}) would be 37. But we note that in both figures the best session is greater than 37. This is because added fitness from error checks are picked up. For example this session, ({cmd 9 ->}, {<- error}, {cmd 0 ->}, {<- ok},

{8 ->}, {<- error}, {0 ->}, {<-ok}, {ksjdf ->}, {<-error}, {calculate ->}, {<-answer}}, would get a basic block (BB) fitness of 43.

Figures 2-5 support the two basic assumptions. First, for programs with VERY few functions, basic blocks did better in terms of percentage of attack surface coverage which translates into how much of the protocol was learned. Figures 2 or 3 could have achieved a maximum fitness of 6 for best session and for total diversity. Yet only 4 of 6 were discovered. This is primarily because of lack of search guidance. (Typically we assume there should be enough functions to effectively discover the protocol. We shall test for this in the next section.) They only learned 2/3 or 66% of a normal session, an abnormal session, and total diversity. Figures 4 and 5 did better. They learned 40+ of 37; over 100% of a “protocol correct” session. That’s because error cases were also tested; this increased fitness.

Though Figures 2 and 3 do not reflect the fitness boost of mixing valid and invalid sessions (creating semi-valid sessions), as in Figures 3 and 4, it is assumed that the GA was still slamming around in a similar fashion. Toward the beginning there’s probably little difference in terms of *unique test cases delivered to target*. However, toward the end of the runs it’s clear that, in this case at least, the BB runs would have been delivering better attack surface test cases.

What about total diversity? Complete code coverage (total diversity) of *TextServer*, while stalking basic blocks, set to *low* is 86. Toward the end of runs Figure 4 is hovering around 45 and Figure 5 is more like 52. That’s 52% and 60%, respectively, of total diversity. How can we achieve better total diversity?

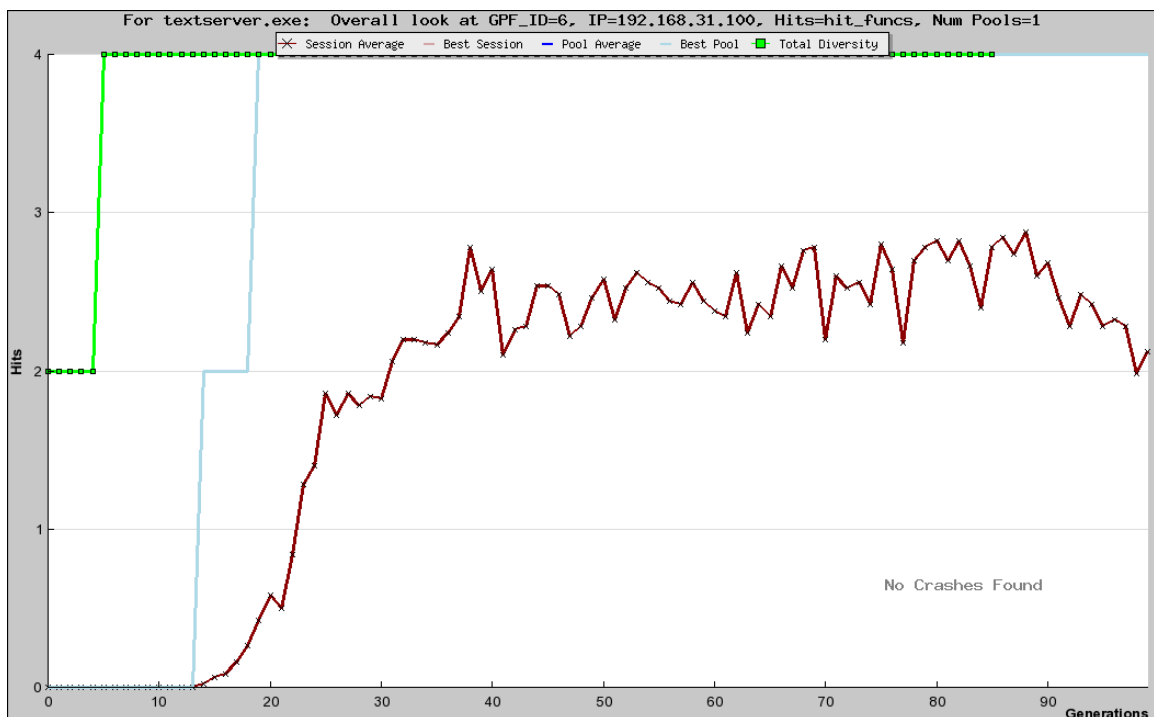


Figure 2: Low, 1 Pools.

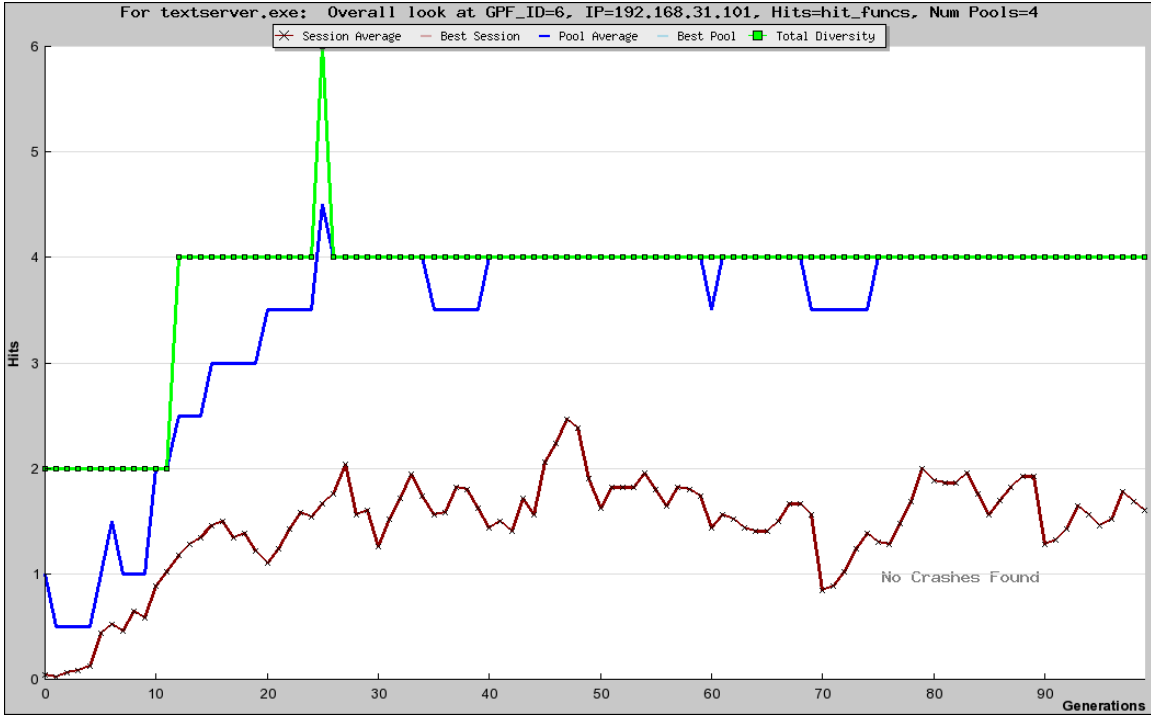


Figure 3: Low, 4 Pools.

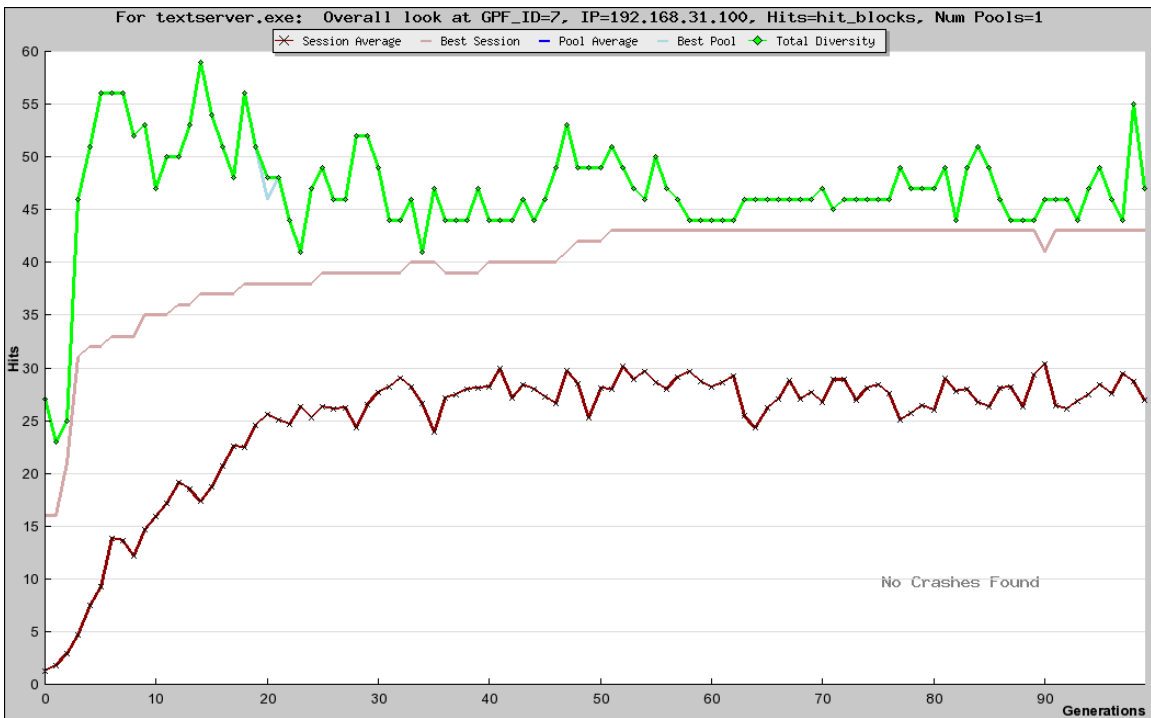


Figure 4: Low. 1 Pool.

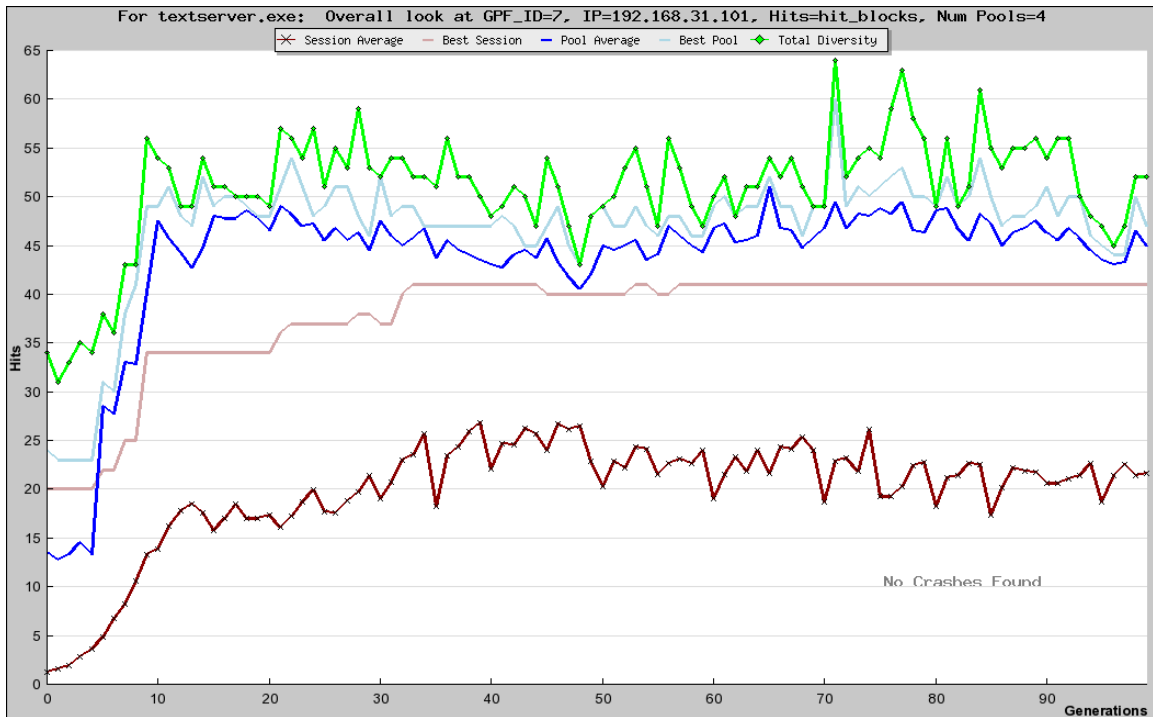


Figure 5: Low. 4 Pools.

## Mommy, where do “Hits” come from?

In this section we detail the origin and nature of the “fitness hits” we continue to reference and use throughout the document.

We begin by showing the calculations of how the fitness numbers work for this section (Figures 6-13). These numbers are only valid for this section as each time the code base for *TextServer* is modified the scaling of these numbers will be changed. (And the code for *TextServer* has since been modified.)

PIDA pre-analysis involves a few steps: Get a validly compiled (non-compressed or encrypted) binary of the test program. Load it into IDApro. Be sure IDA’s disassemble runs with few or no errors. Run the *pida\_dump.py* included in the PaiMei Reverse Engineering toolkit code. Load the PIDA into PaiMei’s *pstalker* module and a message like this will be printed to screen:

```
289 total funcs: 597 total BB's.
```

This indicates that the *TextServer.exe.pida* has the potential to stalk on 289 functions or 597 basic blocks. Not all functions or basic blocks are reachable. See Appendix Figure B for a list of functions discovered by IDApro.

Upon starting a *stalk* (function or basic block runtime tracking) a message such as this will be printed:

```
[*] Filtering 68 points from target id:1 tag id:52
```

[\*] Setting 146 breakpoints on functions in main module

[\*] Filtering 168 points from target id:1 tag id:59

[\*] Setting 430 breakpoints on basic blocks in main module

The above shows the start of a *func* (function) stalk, and then a *BB* (basic block) stalk. Notice for functions the total doesn't add up to 289. Again, this is because we will not be able to reach all of the functions within an executable. Also notice, we had created a filter to ignore breakpoints that will be common to each connection. This is done to speed up EFS. The process to create this type of filter is:

1. Define a filter tag called, "Text\_startup\_conn\_junk\_disconn\_shutdown"
2. Stalk with that tag and record hits to the PaiMei database
3. Start the target application
4. Using *netcat*, connect to the target application
5. Send a few random characters
6. Disconnect
7. Shutdown the target application

Before connecting the GPF portion of EFS, always use this filter tag. Separate filter tags for funcs and BBs (and each new build of *TextServer*) are required.

With the filter in place, the maximum fitness a protocol valid session could achieve is 6 function hits or 36 basic block hits. Just as before, with two error conditions (can't reach third error with less than 10 legs, because there is a maximum errors/session of 10 defined in *TextServer*), would could achieve 6 funcs or 43 BBs.

The maximum fitness a pool (total diversity) could achieve, for the medium setting, would be: 22 funcs or 88 BBs for completely valid sessions or 22 funcs and 94 BBs for semi-valid sessions (picking up all the error conditions).

Here is some further detail of how these hits break down:

- Startup and shutdown = 137 BBs or  $137/597 = 23\%$  of code.
- Network code = 15 BBs or  $15/597 = 3\%$  of code
- **Parsing** = 94 BBs or  $16\%$  of code. *This is the portion of code likely to contain bugs!*
- Total Attack surface = network code + parsing. 109bb or  $18\%$  of code.
- Code accounted for: 137+94bb or  $39\%$ . (68+22funcs or 31%)

Notice, via attack surface testing (in this case external network) we cannot account for or reach much of the code. Yet, EFS sets breakpoints on all functions within an executable, regardless of whether or not they are actually used. This is fine and does not affect the performance of EFS. However, it's important

to keep in mind that looking at a percentage of code coverage from a gray-box perspective is only so-so useful. Also note that it's **impossible** to achieve 100% coverage of all the application code when testing an interface such as the remote attack surface. If percentage calculations are desirable, only the interface under test should be considered.

## Running TextServer on Medium

This section shows that learning via functions is sufficient, and begins the investigation of diversity within pools.

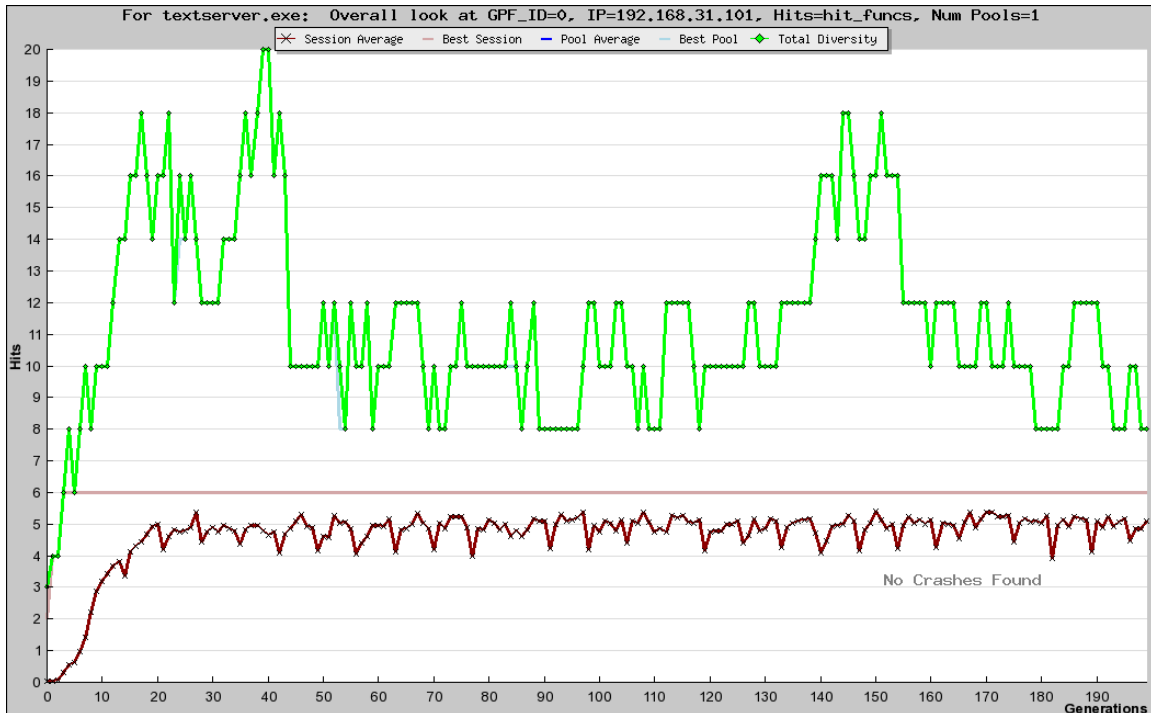


Figure 6: Med, 1 Pool

Understanding Figure 6 is pretty simple. The best a session can do is 6 hit functions. We see the pink *Best Session* line steady right at 6 from almost the beginning. The red line with x's, *Average Sessions*, approach that, but will never reach 6. This is because there are always some sessions in the pools that are not as developed as the best session. The *Pool Average* (blue), *Best Pool* (light blue), and *Total Diversity* (green with diamonds) are the same because there is only one pool. (Note that we only see one line because if two lines are in the same position *Jpgraph* only displays the last drawn.)

The fact that Figure 6 was able to quickly learn the best session shows that with more hit possibilities EFS is able to effectively learn a simple (very forgiving) protocol using only functions. This should be reinvestigated for the *BinaryServer* (i.e. less forgiving protocols). While it's true that the best session is 6 for *low*, *med*, or *high*, when running *TextServer* on *med* or *high* there are more total ways (combinations) that a best can be found. In other words, diversity becomes an

option when the server is set to medium or high. For the low setting, the exact session ({cmd 0 ->}, {<- ok}, {0 ->}, {<-ok} {calculate ->}, {<-answer}) had to be discovered. But now any x, 0-4, could replace the 0 in that session. For example, ({cmd 3 ->}, {<- ok}, {1 ->}, {<-ok} {calculate ->}, {<-answer}) would also find the “best session”. The strings (cmd, 0, 1, etc.) could be found randomly or are sometimes drawn from the EFS seedfile. For a deeper understand of how EFS works see [1]. See Appendix Figure C for the *TextServer* experiments seedfile.

Also, in Figure 6 we see the total pool diversity jumping around but it should ultimately move down toward the best session, as that session dominates more and more of the pool. The diversity bobs up and down because there are 9 different ways a session can be the best. As it does toward the end, it is expected to level out to approximately the best session. This would be even more obvious if EFS found one path that was much better than the rest, but in this case all 9 are equal. Again, since bugs could lie down these less fit alternatives we wish to foster diversity among sessions.

The 4 pool test was run twice (Figures 7 and 8) because it’s difficult to decipher the results. Figure 8 appears to have an upward trend, but Figure 7 looks flat. We note a total diversity spike of 20 in the single pool, and neither 4 pool can match that. The final diversity average looks better in Figure 7 and about the same in Figure 6. Note that the session average is less with multiple pools. That indicates less total convergence toward the single best session, which should be considered a positive trait in terms of diversity. It could also mean that if the protocol to discovery is complex it will take longer.

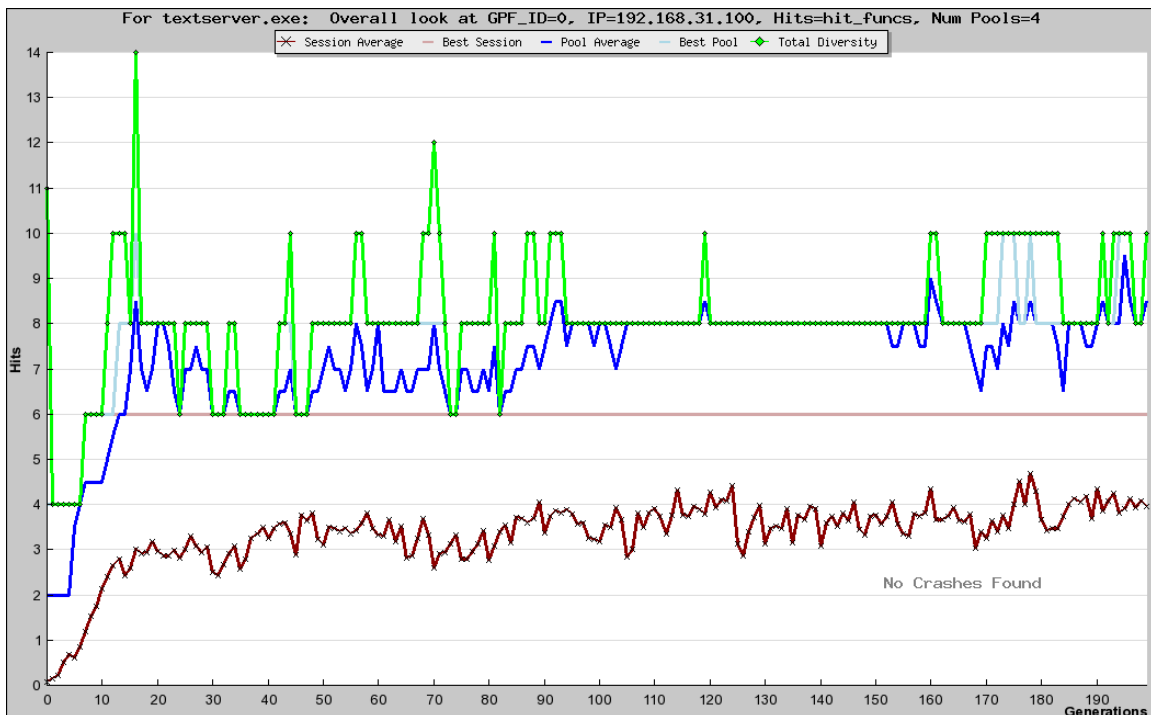


Figure 7: Med, 4 Pools (host b)

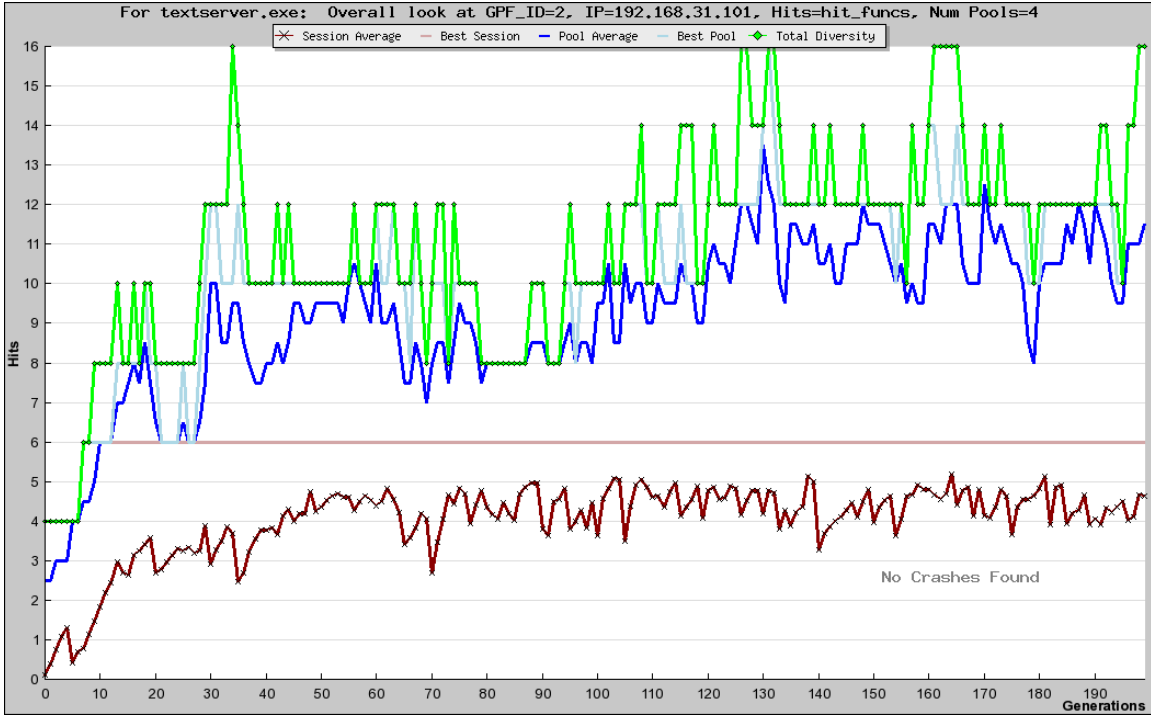


Figure 8: Med, 4 Pools

The run with 10 pools (Figure 9) is equally difficult to decipher. Is there a diversity benefit by running multiple pools or not? With less converged sessions and no downward diversity trend it seems there is, but it's not blaringly obvious.

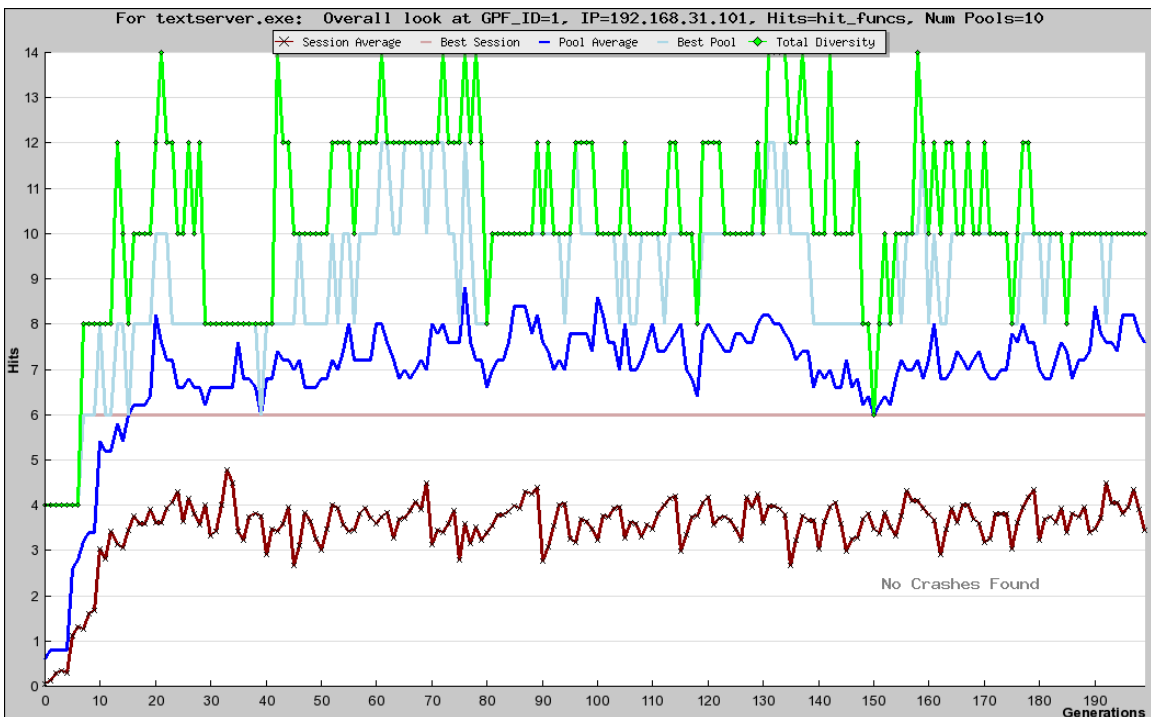


Figure 9: Med, 10 Pools

### Different Crossover Rates and a Switch to *High*

In this section we changed the hit type to basic blocks. Figure 10 shows what was expected: a downward diversity trend more pronounced than before. Figure 11 does have a small upward trend in terms of diversity.

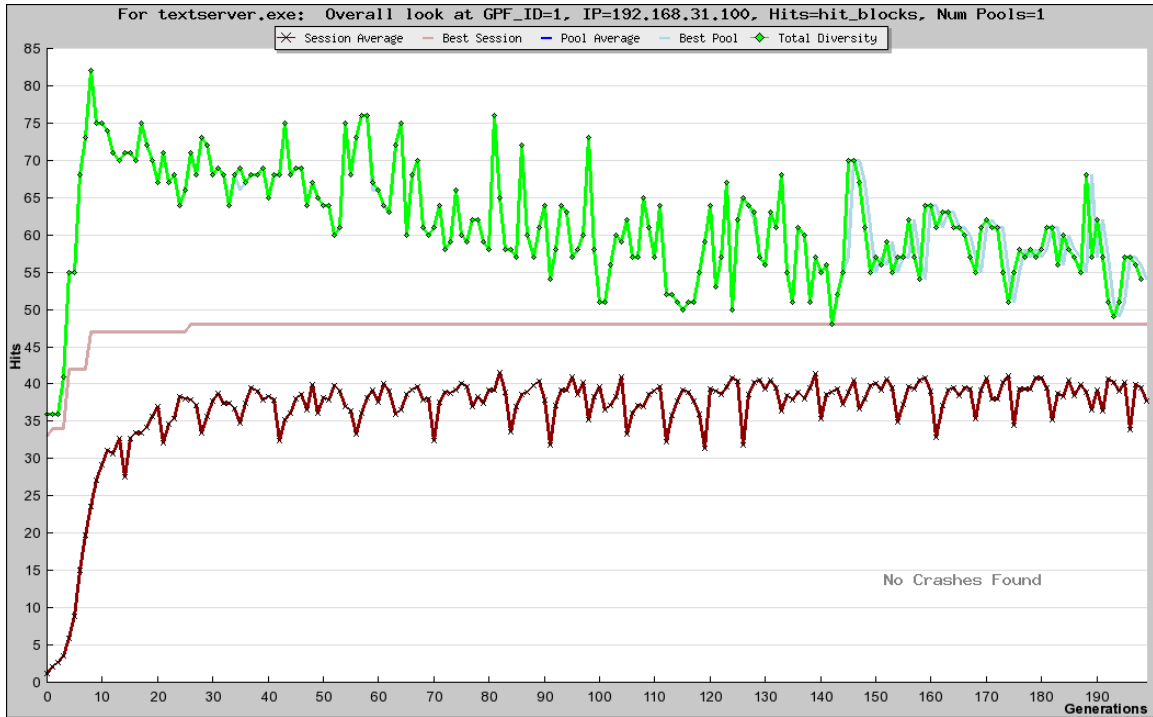


Figure 10: Med, 1 Pool BBs

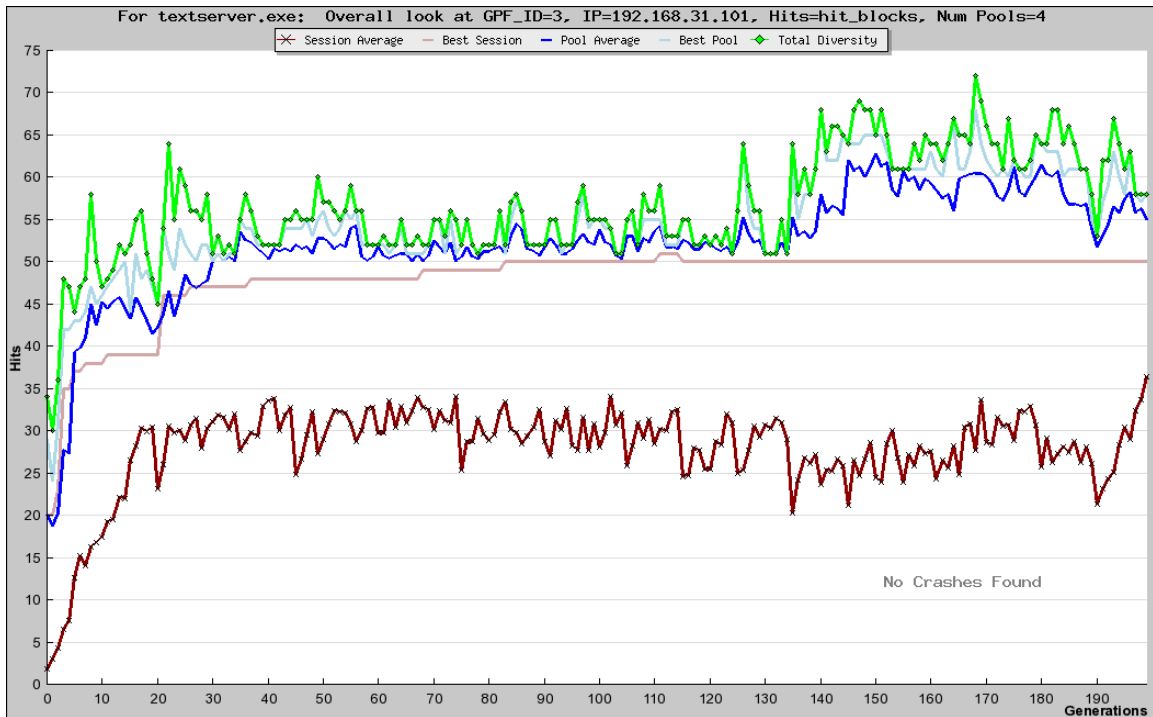


Figure 11: Med, 4 Pools BBs

Since the benefit of multiple pools still isn't blindly obvious, session and pool mutation, and pool crossover are removed (Figure 12 and 13). By not allowing pools to breed EFS seems to maintain a bit more total diversity in Figure 13. This makes sense, because when pools breed the same thing happens when sessions breed. The strongest will survive, and ultimately dominate. But this doesn't prove that multiple pools are better. Also, note that without pool breeding the best pool will likely falter down to simply the best sessions. One interesting thing though, normally total diversity tracks closely with best pool, but by not allowing pools to breed the two seem to have separated.

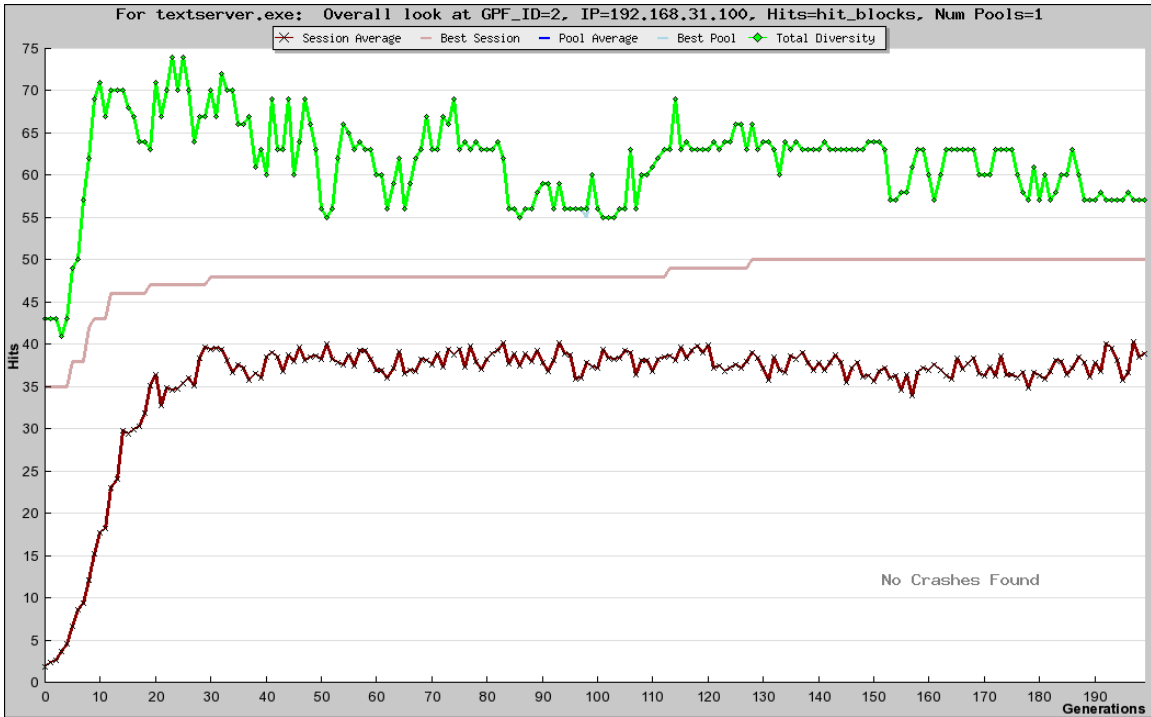


Figure 12: Med, 1 Pool. No Session Mutation or Pool Crossover/Mutation

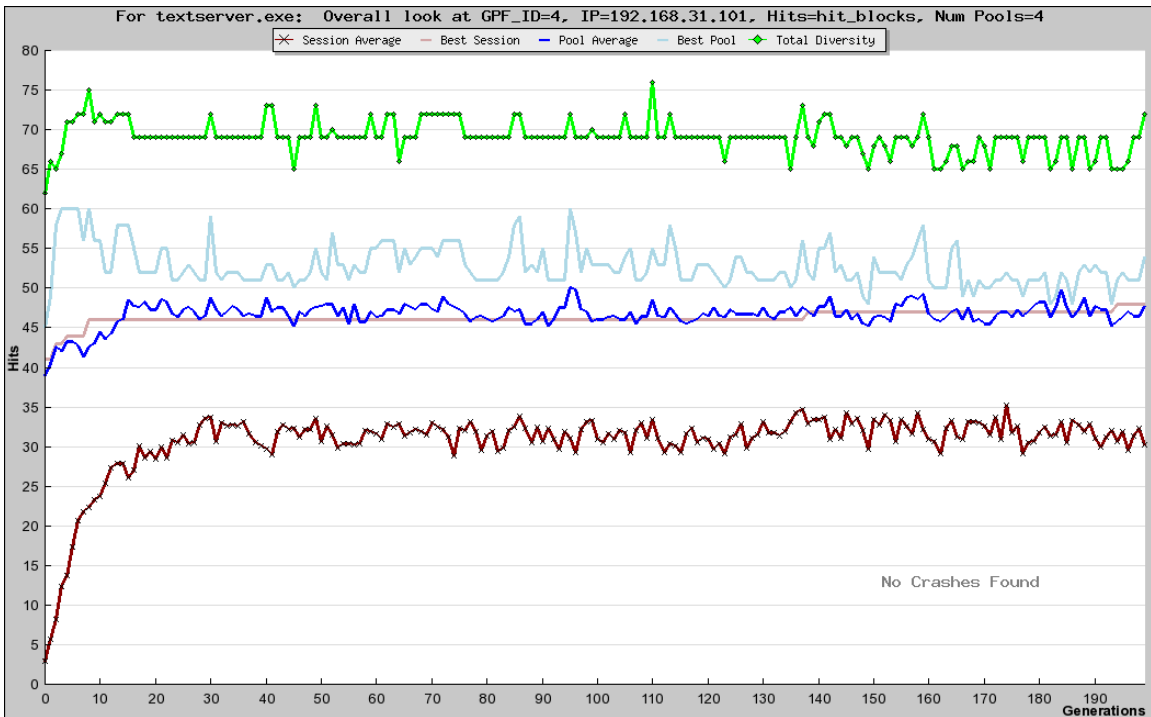


Figure 13: Med, 4 Pools. No Session Mutation or Pool Crossover/Mutation

In Figures 14-17 we reintroduce mutation and crossover and run *TextServer* on *high* (19 paths).

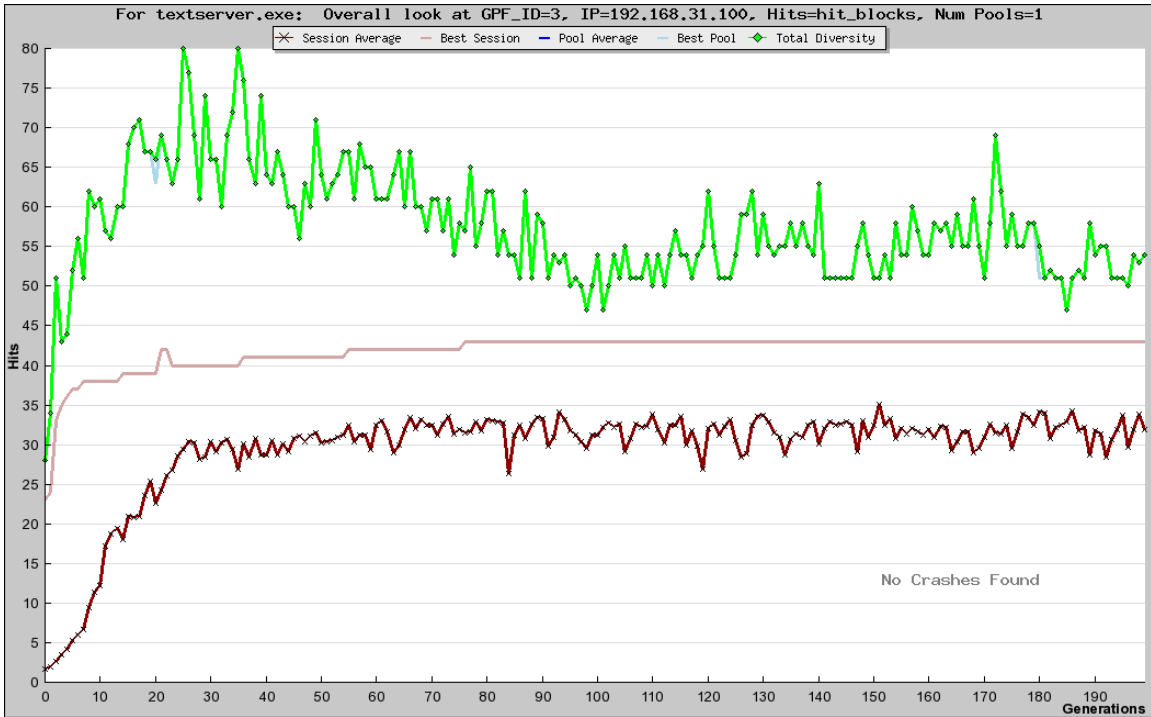


Figure 14: High, 1 Pool.

As expected, Figure 14 looks a lot like Figure 10. On high, there's an opportunity to cover more paths, and to small extent this happens.

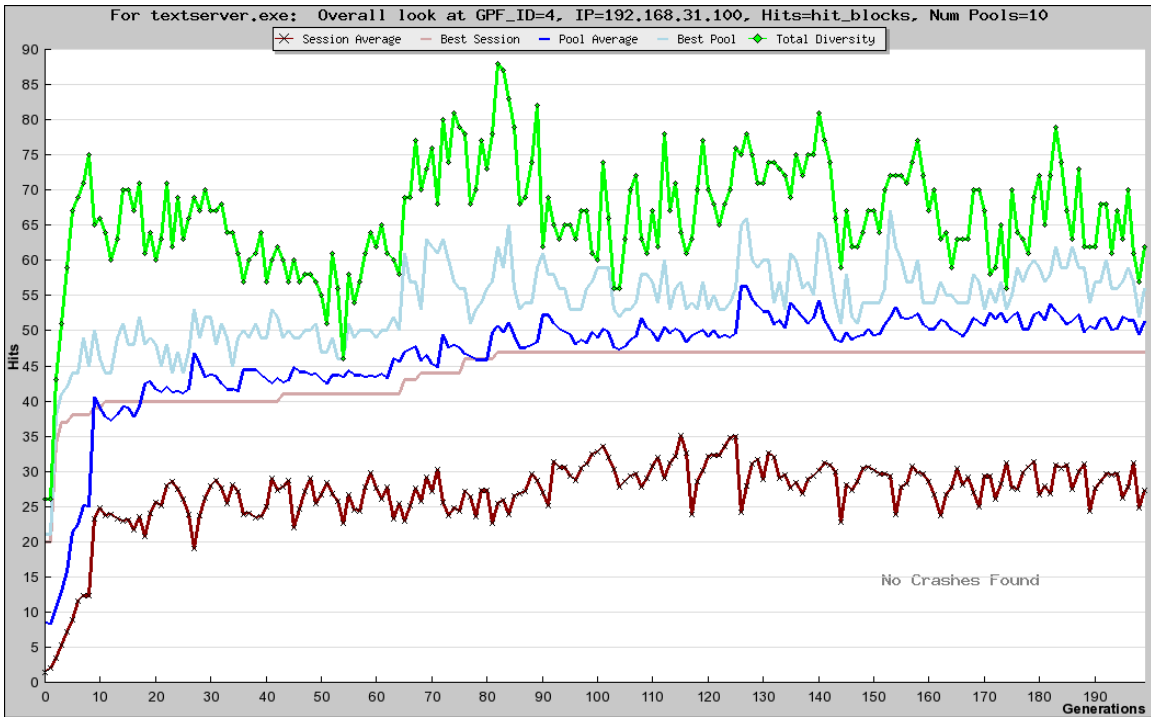


Figure 15: High, 10 Pools. (9 Pool crossover, 13 Pool Mutation)

Figure 15 is encouraging; we see a stronger total diversity throughout the run.

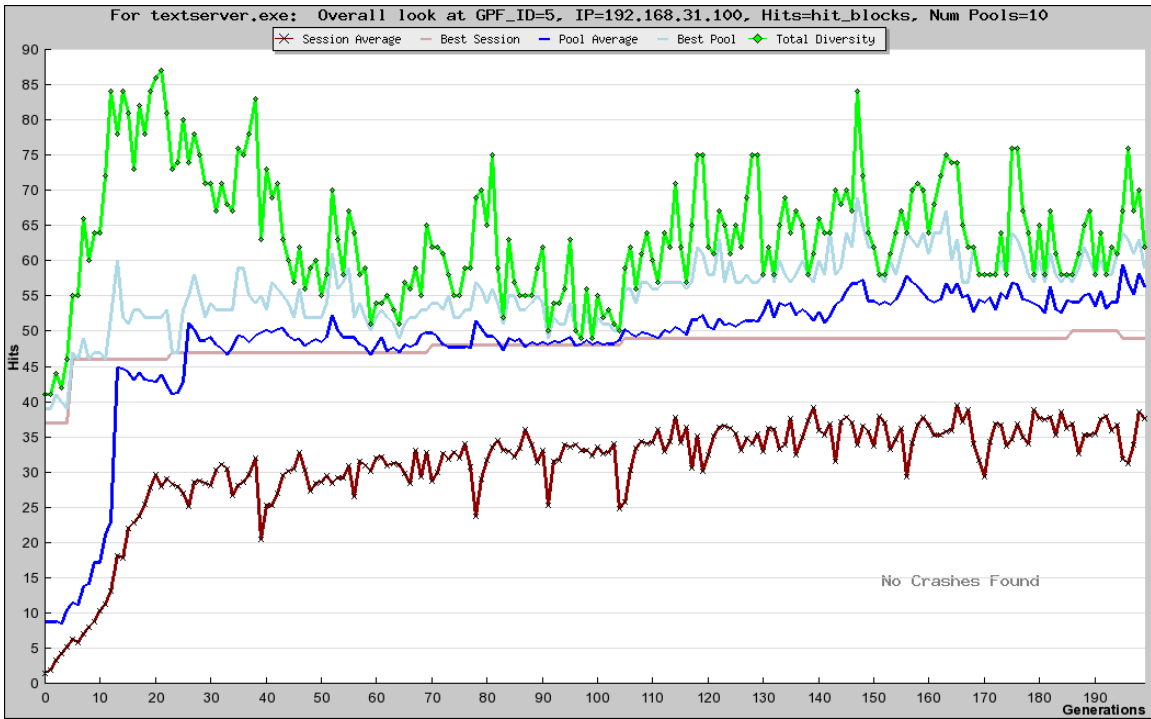


Figure16: High, 10 Pools. (13 Pool crossover, 17 Pool Mutation)

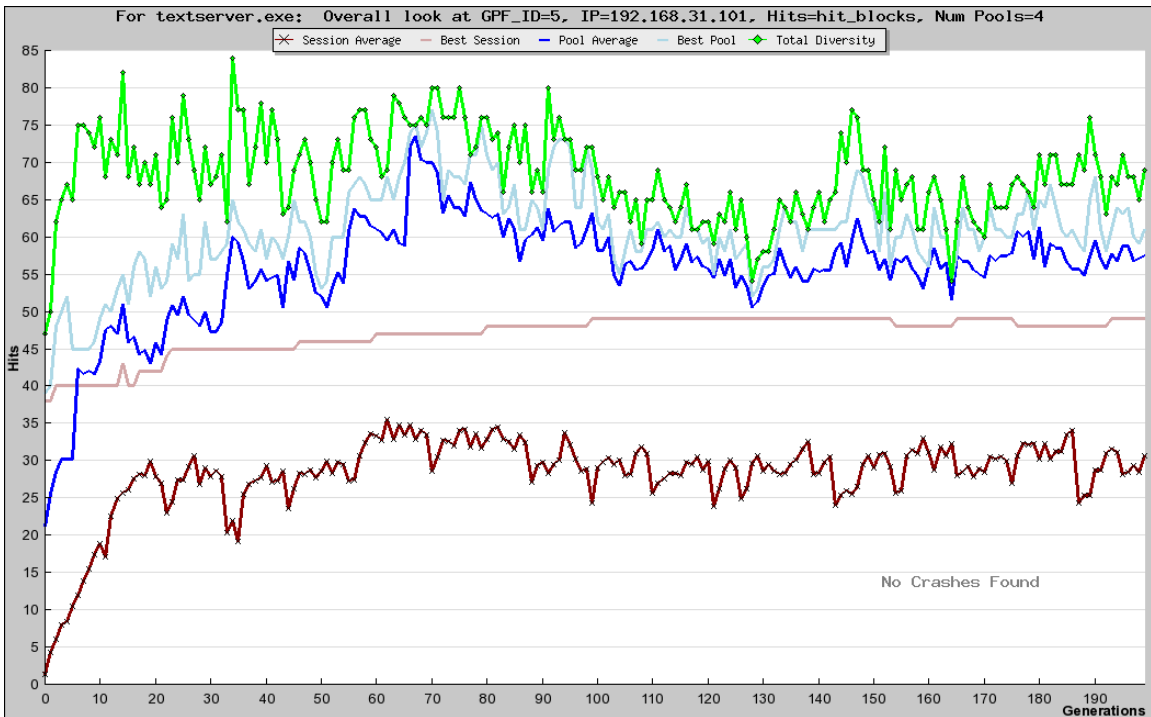


Figure17: High, 4 Pools. (11 Pool crossover, 15 Pool Mutation)

In Figures 15, 16, and 17 the pool crossover and mutation parameters were toyed with to see if a major difference would be observed. No obvious benefit is noticed.

In fact, the total possible diversity for *high* is 150 BBs. Considering that none of the multiple pools run observed achieve that, the niching option should be explored.

But first, can EFS learn a binary protocol? The previous protocol is text, which translates into very forgiving. For example, if we send “lkjsdfilkjsdfj cmd 1 \r\n dfg g fdg dfg cmd 6 lkjsdfjlsdfjlsdfjlsfjsdfjlsdkjf” the text server will parse as “cmd 1\r\n”. Structured protocols are not so generous.

### **Binary Protocol**

Figure X shows the binary protocol structure.

Client Request Message Structure →:

Total LEN 4 bytes	Session ID 4 bytes	CMD LEN 2 bytes	CMD Str Var bytes
----------------------	-----------------------	--------------------	----------------------

← Server Response Message Structure:

Total LEN 4 bytes	Session ID 4 bytes	RSP LEN 2 bytes	RSP Str Var bytes
----------------------	-----------------------	--------------------	----------------------

Figure 18: The Protocol for *BinaryServer*

This protocol is similar to the text protocol in that there are 19 paths on *high*. However, stricter rules regarding message structure must be followed: The lengths must be correct. The session ID chosen to begin with must be used throughout a sequence of commands.

### **Niching, another way to foster diversity**

Pools seem beneficial, but it's prudent to try another approach to maintain session diversity. A new fitness function of the following form is applied to the sessions and pools after the hit and diverse hits have been calculated. This approach can be used in conjunction with multiple pools as it can calculate a diversity boost for each pool as well.

$$\text{Fitness} = \text{Hits} + ( (\text{UNIQUE}/\text{BEST}) * (\text{BEST}-1) )$$

Total fitness for the below example sessions would be:

$$A = 7 \text{ hits} + ( 0/7 * 6 ) = 7 \text{ total fitness}$$

$$B = 4 \text{ hits} + ( 4/7 * 6 ) = 7.4 \text{ total fitness}$$

$$C = 5 \text{ hits} + ( 2/7 * 6 ) = 6.7 \text{ total fitness}$$

$$D = 6 \text{ hits} + ( 0/7 * 6 ) = 6 \text{ total fitness}$$

$$E = 1 \text{ hit} + ( 1/7 * 6 ) = 1.9 \text{ total fitness}$$

Notice that in rare cases it's possible for a session to have few hits than the original best, but end up to be rated as the best session. This will result in that diverse session being automatically carried over to the next generation due to the elitism of 1. It's unclear if this is ideal, but it seems to be the right approach as fostering diversity is very important. Likely the original best will be maintain in part since it's likely saturated the session already. The only situation where this less fit session being kept over an originally more fit session would be at the very start of a run; it could delay the learning process if the hit best is lost entirely. This situation seems unlikely, but tests will be conducted. Figure XX and XX shows the diversity of a single pool run using function hits on *high*. They are very different, which shows the unpredictable nature of GAs. The first did unusually poor, but this can happen. The GA did not happen to select or find the proper seeds or combination of terms to allow it to discover a full session. Further runs show the chances of this happening are low. (gpfid 2,3)

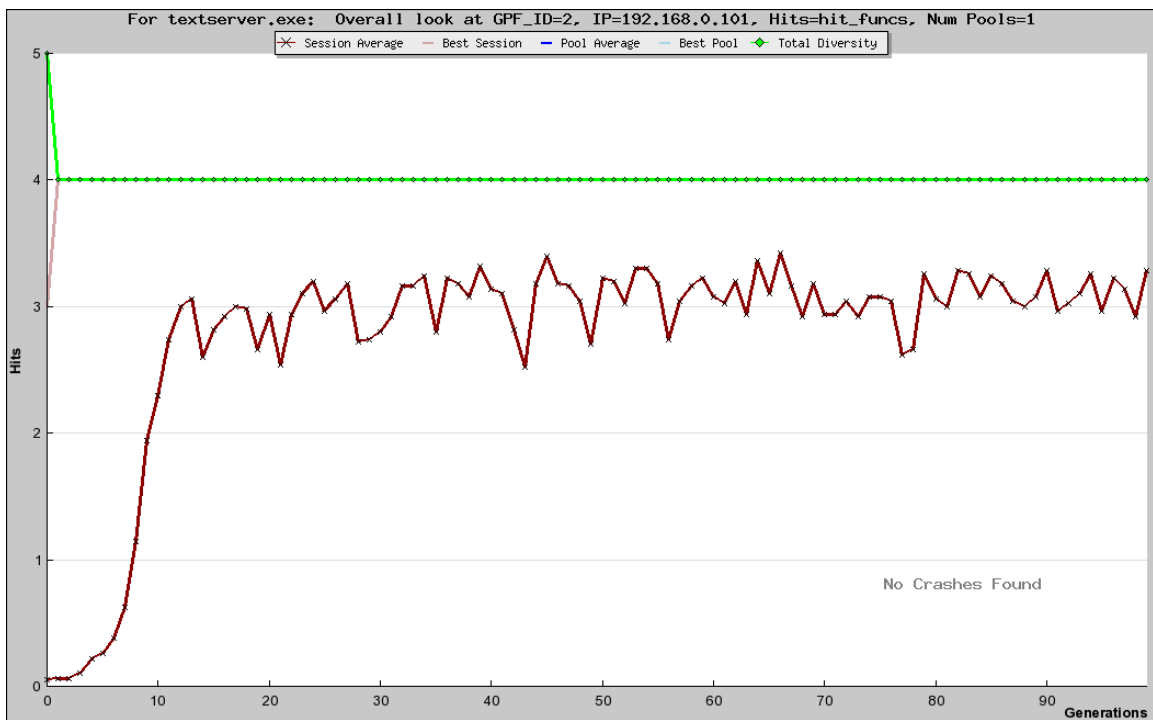


Figure XX: 1 Pool, Funcs, High, Diversity

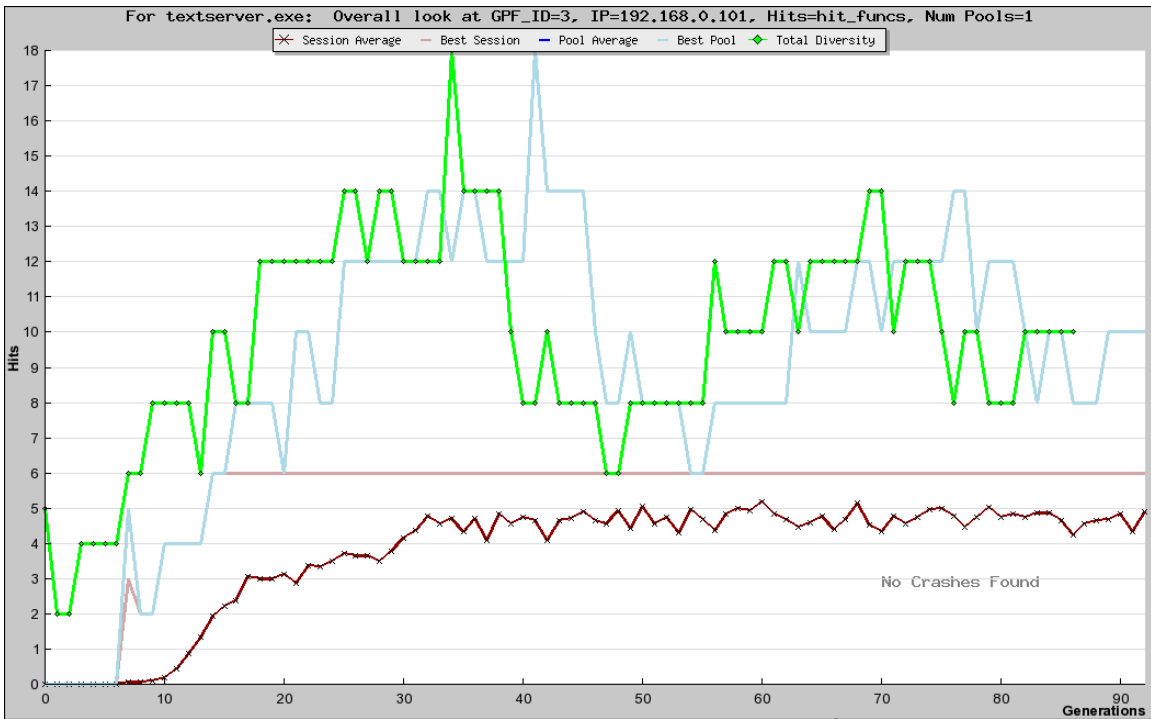


Figure XX: 1 Pool, funcs, high, diversity (2<sup>nd</sup> run)

Figure XX used 4 pool and funcs while on *high*. (gpfid 4)

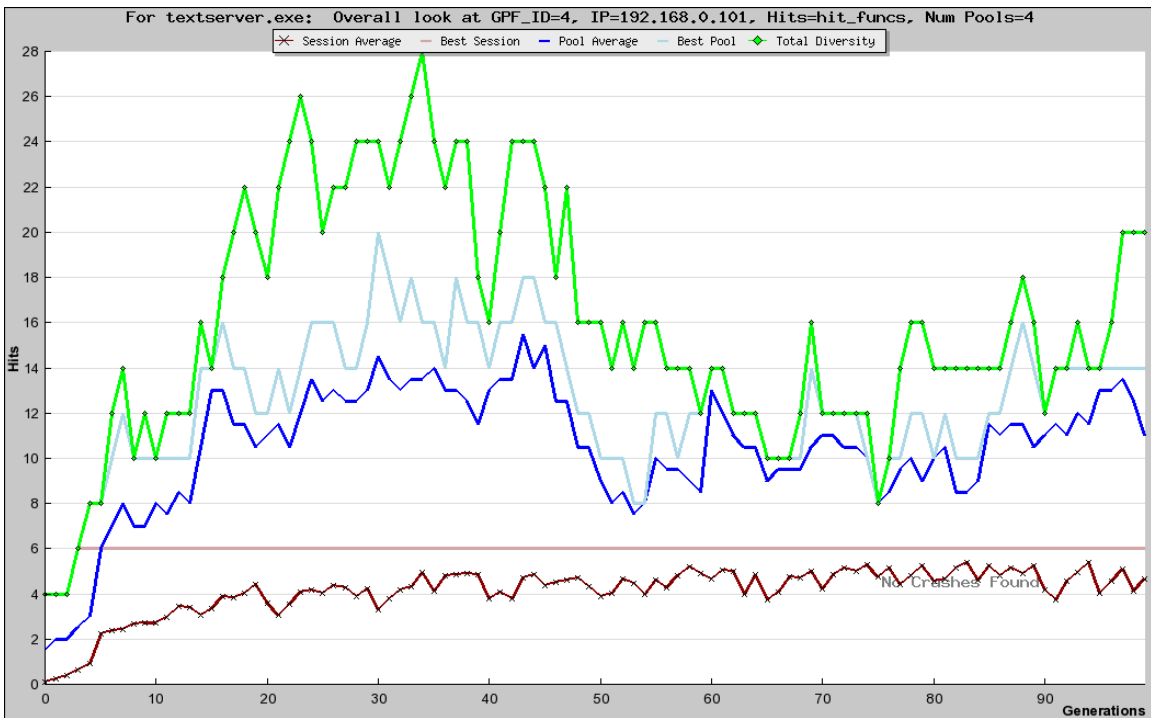


Figure XX: 4 Pools, funcs, high, diversity

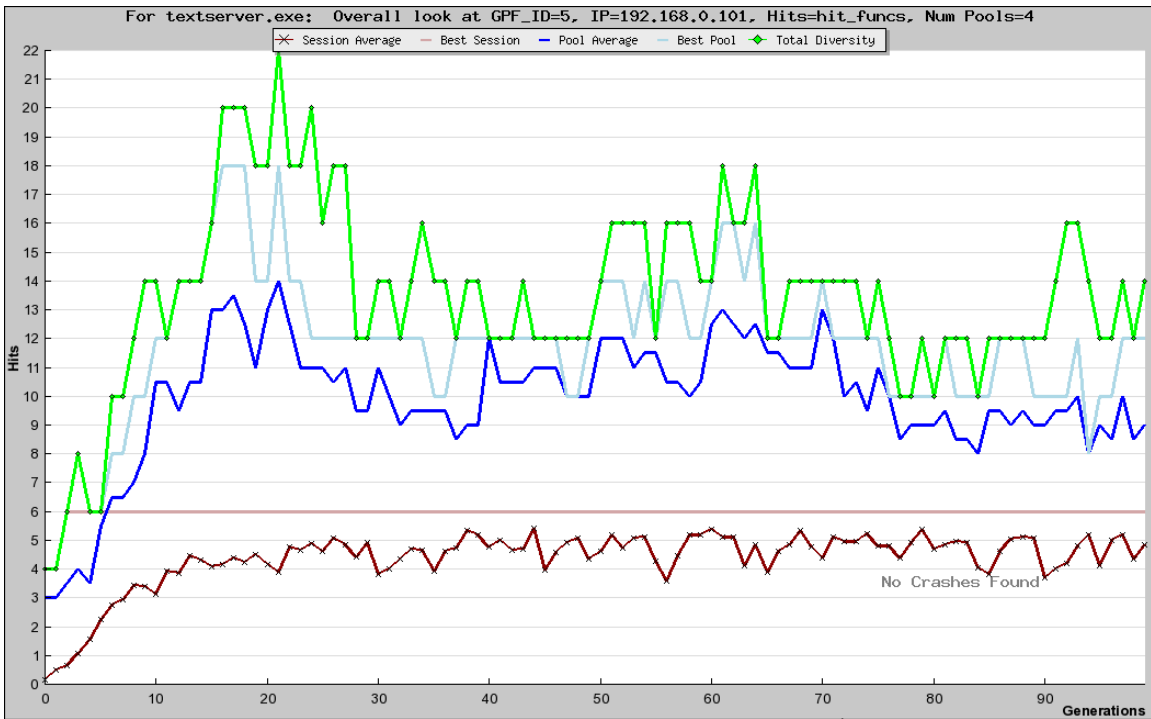


Figure XX: 4 Pools, funcs, high, diversity (2<sup>nd</sup> run)

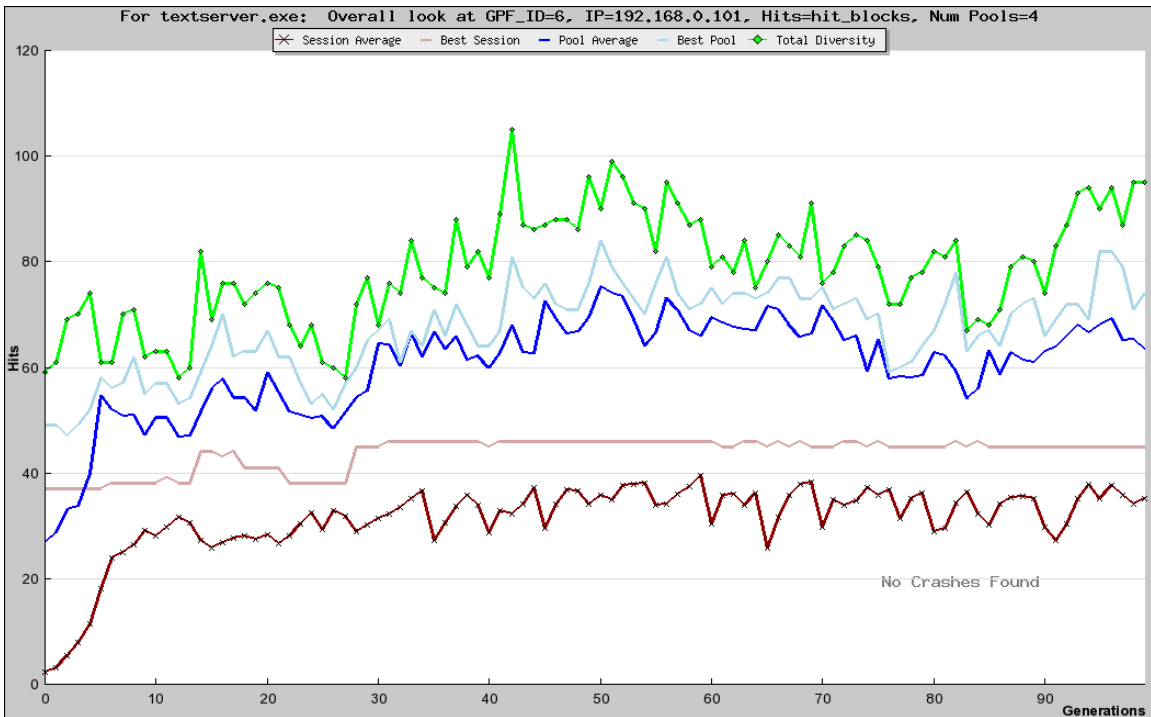


Figure XX used 1 pool and basic blocks on *high*. Figure XX shows 4 pools and basic blocks while run in *high* mode.

## ***Errors in the Text Protocol (haven't implemented these two sections at all ...)***

There are four two bugs. The first is a stack overflow in the password field; any password longer than 256 bytes. The second is a heap overflow in any "cmd x". That is, cmd + longstring, will result in *delayed* heap corruption. The third bug is an off-by-one error in the "cmd 2 2y\r\n". If something such as "cmd 2 2y\r\n" is supplied the error will be triggered. The forth error is a string format bug in "cmd 3 y". If y is replaced with something like %n the bug is triggered.

## ***Errors in the Binary Protocol***

### **References:**

- [1] J. DeMott, R. Enbody, W. Punch, "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing", *BlackHat and Defcon 2007*
- [2] P. McMinn and M. Holcombe, "Evolutionary Testing Using an Extended Chaining Approach", *ACM Evolutionary Computation*, Pgs 41-64, Volume 14, Issue 1 (March 2006)

### Appendix:

App	Func or BB	GPF ID	Test Host	Pools	Sess	Fixed or Max	Legs	Fixed or Max	Toks	Fixed or Max	Total Gens	Sess Mutation	Pool Cross	Pool Mut	Paths	Fig
Text	func	0	a	1	100	Fixed	10	Fixed	5	Fixed	200	7	5	9	Med	6
Text	func	0	b	4	25	Fixed	10	Fixed	5	Fixed	200	7	5	9	Med	7
Text	func	1	a	10	10	Fixed	10	Fixed	5	Fixed	200	7	5	9	Med	9
Text	func	2	a	4	25	Fixed	10	Fixed	5	Fixed	200	7	5	9	Med	8
Text	bb	1	b	1	100	Fixed	10	Fixed	5	Fixed	200	7	5	9	Med	10
Text	bb	3	a	4	25	Fixed	10	Fixed	5	Fixed	200	7	5	9	Med	11

Text	bb	2	b	1	100	Fixed	10	Fixed	5	Fixed	200	200	200	200	Med	12
Text	bb	4	a	4	25	Fixed	10	Fixed	5	Fixed	200	200	200	200	Med	13
Text	bb	3	b	1	100	Fixed	10	Fixed	5	Fixed	200	7	5	9	High	14
Text	bb	5	a	4	25	Fixed	10	Fixed	5	Fixed	200	7	11	15	High	17
Text	bb	4	b	10	10	Fixed	10	Fixed	5	Fixed	200	7	9	13	High	15
Text	bb	5	b	10	10	Fixed	10	Fixed	5	Fixed	200	7	13	17	High	16
Text	func	6	a	4	25	Fixed	10	Fixed	5	Fixed	100	7	5	9	Low	2
Text	func	6	b	1	100	Fixed	10	Fixed	5	Fixed	100	7	5	9	Low	3
Text	bb	7	a	4	25	Fixed	10	Fixed	5	Fixed	100	7	5	9	Low	4
Text	bb	7	b	1	100	Fixed	10	Fixed	5	Fixed	100	7	5	9	Low	5

Appendix Figure A: Primary EFS Parameter Settings for Figures 2-16

```

j__setdefaultprecision .text 00411005 00000005 R . . . . .
j__six .text 0041100A 00000005 R . . . . .
j__seven .text 0041100F 00000005 R . . . . .
j__setargv .text 00411014 00000005 R . . . . .
j__GetFinal .text 0041102D 00000005 R . . . . .
j__eight .text 00411032 00000005 R . . . . .
j__security_check_cookie .text 0041103C 00000005 R . . . . .
j__std__basic__ostream__char__std__char__traits__char____sentry__operator__bool .text 00411046 00000005 R . . . . .
j__five .text 00411055 00000005 R . . . . .
j__RTC_GetErrorFuncW .text 00411064 00000005 R . . . . .
j__zero .text 00411073 00000005 R . . . . .
j__two .text 00411082 00000005 R . . . . .
j__security_init_cookie .text 00411091 00000005 R . . . . .
j__atexit .text 004110AA 00000005 R . . . . .
j__memset .text 004110BE 00000005 R . . . . T .
j__lock .text 004110C8 00000005 R . . . . .
j__RTC_CheckStackVars .text 004110DC 00000005 R . . . . .
j__initterm .text 004110FF 00000005 R . . . . .
j__strlen .text 00411104 00000005 R . . . . T .
j__strcmp .text 00411118 00000005 R . . . . T .
j__FindPESection .text 0041113B 00000005 R . . . . .
j__four .text 00411145 00000005 R . . . . .
j__GetSubOption .text 00411159 00000005 R . . . . .
j__RTC_StackFailure .text 0041115E 00000005 R . . . . .
j__crt_debugger_hook .text 0041116D 00000005 R . . . . .
j__ValidatelImageBase .text 00411172 00000005 R . . . . .
j__three .text 0041117C 00000005 R . . . . .
j__std__basic__ostream__char__std__char__traits__char____sentry__sentry .text 0041118B 00000005 R . . . . .
j__RTC_SetErrorFuncW .text 0041119A 00000005 R . . . . .
j__onexit .text 0041119F 00000005 R . . . . T .
j__NtCurrentTeb .text 004111A4 00000005 R . . . . .
j__std__operator__std__char__traits__char____ .text 004111B3 00000005 R . . . . .
j__invoke_watson_if_error .text 004111BD 00000005 R . . . . .
j__std__basic__ostream__char__std__char__traits__char____Sentry_base____Sentry_base .text 004111C2 00000005 R . . . . .
. . .
j__Server .text 004111D1 00000005 R . . . . .
start .text 004111D6 00000005 R . . . . .
j__unlock .text 004111EA 00000005 R . . . . .
j__HandleConnection .text 004111FE 00000005 R . . . . .
j__RTC_CheckEsp .text 00411203 00000005 R . . . . .
j__main .text 00411208 00000005 R . . . . .
j__RTC_Initialize .text 0041120D 00000005 R . . . . .
j__controlfp_s .text 0041121C 00000005 R . . . . .
j__invoke_watson .text 0041122B 00000005 R . . . . .
j__RTC_GetSrcLine .text 00411230 00000005 R . . . . .

```

```

j__CRT_RTC_INITW                .text 00411235 00000005 R . . . . .
j__one                          .text 0041123F 00000005 R . . . . .
j__IsNonwritableInCurrentImage  .text 00411249 00000005 R . . . . .
j__amsg_exit                    .text 00411258 00000005 R . . . . .
j__XcptFilter                   .text 0041125D 00000005 R . . . . .
j__std__basic_ostream_char_std__char_traits_char____sentry__sentry      .text 00411280 00000005 R . . . . .
j__GetCommandNumber            .text 00411285 00000005 R . . . . .
j__except_handler4_common      .text 0041128A 00000005 R . . . . .
j__RTC_Failure                 .text 004112A3 00000005 R . . . . .
j__RTC_AllocaFailure           .text 004112AD 00000005 R . . . . .
j__std__basic_ostream_char_std__char_traits_char____Sentry_base__Sentry_base .text 004112BC 00000005 R . . . . .
. . .
j__dllonexit                    .text 004112C1 00000005 R . . . . .
j__initterm_e                  .text 004112CB 00000005 R . . . . .
j__RTC_GetErrorFunc           .text 004112D5 00000005 R . . . . .
GetCommandNumber              .text 004115B0 0000031F R . . . B T .
GetSubOption                 .text 004119C0 00000257 R . . . B T .
zero                          .text 00411CD0 0000004E R . . . B T .
one                           .text 00411D40 0000004E R . . . B T .
two                           .text 00411DB0 0000004E R . . . B T .
three                         .text 00411E20 0000004E R . . . B T .
four                          .text 00411E90 0000004E R . . . B T .
five                          .text 00411F00 0000004E R . . . B T .
six                           .text 00411F70 0000004E R . . . B T .
seven                         .text 00411FE0 0000004E R . . . B T .
eight                        .text 00412050 0000004E R . . . B T .
GetFinal                     .text 004120C0 000001AC R . . . B T .
HandleConnection            .text 00412310 00000232 R . . . B T .
Server                       .text 00412620 00000227 R . . . B . .
main                          .text 00412950 0000012A R . . . B . .
std__operator__std__char_traits_char____ .text 00412AD0 0000043C R . . . B . .
std__basic_ostream_char_std__char_traits_char____sentry__sentry      .text 00413040 00000112 R . . . B . .
std__basic_ostream_char_std__char_traits_char____sentry__sentry      .text 004131A0 000000A1 R . . . B . .
std__basic_ostream_char_std__char_traits_char____sentry__operator_bool .text 00413270 00000030 R . . . B . .
std__basic_ostream_char_std__char_traits_char____Sentry_base__Sentry_base .text 004132B0 00000095 R . . . B . .
std__basic_ostream_char_std__char_traits_char____Sentry_base__Sentry_base .text 00413370 00000088 R . . . B . .
send(x,x,x,x)                 .text 0041341A 00000006 R . . . . T .
closesocket(x)               .text 00413420 00000006 R . . . . T .
recv(x,x,x,x)                 .text 00413426 00000006 R . . . . T .
WSACleanup()                 .text 0041342C 00000006 R . . . . T .
inet_ntoa(x)                  .text 00413432 00000006 R . . . . T .
accept(x,x,x)                 .text 00413438 00000006 R . . . . T .
listen(x,x)                   .text 0041343E 00000006 R . . . . T .
bind(x,x,x)                   .text 00413444 00000006 R . . . . T .
socket(x,x,x)                 .text 0041344A 00000006 R . . . . T .
htons(x)                      .text 00413450 00000006 R . . . . T .
WSAStartup(x,x)              .text 00413456 00000006 R . . . . T .
std::basic_ios<char,std::char_traits<char>>::setstate(int,bool) .text 0041345C 00000006 R . . . . .
std::ios_base::width(int)      .text 00413462 00000006 R . . . . .
std::basic_streambuf<char,std::char_traits<char>>::sputc(char const *,int) .text 00413468 00000006 R . . . . .
std::char_traits<char>::eq_int_type(int const &,int const &) .text 0041346E 00000006 R . . . . .
std::char_traits<char>::eof(void) .text 00413474 00000006 R . . . . .
std::basic_streambuf<char,std::char_traits<char>>::sputc(char) .text 0041347A 00000006 R . . . . .
std::basic_ios<char,std::char_traits<char>>::rdbuf(void) .text 00413480 00000006 R . . . . .
std::basic_ios<char,std::char_traits<char>>::fill(void) .text 00413486 00000006 R . . . . .
std::ios_base::flags(void) .text 0041348C 00000006 R . . . . .
std::ios_base::width(void) .text 00413492 00000006 R . . . . .
std::char_traits<char>::length(char const *) .text 00413498 00000006 R . . . . .
std::basic_ostream<char,std::char_traits<char>>::flush(void) .text 0041349E 00000006 R . . . . .
std::basic_ios<char,std::char_traits<char>>::tie(void) .text 004134A4 00000006 R . . . . .
std::ios_base::good(void) .text 004134AA 00000006 R . . . . .
std::basic_ostream<char,std::char_traits<char>>::_Osfx(void) .text 004134B0 00000006 R . . . . .
std::uncaught_exception(void) .text 004134B6 00000006 R . . . . .
std::basic_streambuf<char,std::char_traits<char>>::_Lock(void) .text 004134BC 00000006 R . . . . .

```

```

std::basic_streambuf<char,std::char_traits<char>>::_Unlock(void) .text 004134C2 00000006 R . . . . .
__stricmp .text 004134C8 00000006 R . . . . T .
__toupper .text 004134CE 00000006 R . . . . T .
__strtok .text 004134D4 00000006 R . . . . T .
_RTC_CheckEsp .text 004134E0 00000025 R . . . B . .
_RTC_CheckStackVars .text 00413510 00000005F R . . . B . .
_RTC_AllocHelper .text 00413590 00000030 R . . . B . .
_RTC_CheckStackVars2 .text 004135D0 000000D5 R . . . B . .
_RTC_InitBase .text 004136E0 00000029 R . . . . .
_RTC_Shutdown .text 00413720 00000013 R . . . . .
__memset .text 00413738 00000006 R . . . . T .
__security_check_cookie .text 00413740 0000000F R . . . . .
__strlen .text 00413752 00000006 R . . . . T .
__sprintf .text 00413758 00000006 R . . . . T .
__exit .text 0041375E 00000006 . . . . T .
__CxxFrameHandler3 .text 00413764 00000006 R . . . . .
pre_c_init .text 00413770 00000099 R . . . B . .
pre_cpp_init .text 00413830 00000058 R . . . B . .
mainCRTStartup .text 004138A0 0000000F R . . . B . .
__tmainCRTStartup .text 004138C0 00000244 R . . . B . .
NtCurrentTeb .text 00413BA0 0000000B R . . . B . .
check_managed_app .text 00413BB0 000000AD R . . . B . .
DebuggerKnownHandle .text 00413C90 0000000E R . . . . .
DebuggerProbe .text 00413CB0 00000086 R . . . B . .
_RTC_Failure .text 00413D60 00000050 R . . . B . .
failwithmessage .text 00413DD0 00000235 R . . . B T .
DebuggerRuntime .text 004140A0 00000098 R . . . B . .
_RTC_StackFailure .text 00414160 0000011F R . . . B . .
_RTC_AllocFailure .text 004142D0 00000131 R . . . B . .
__getMemBlockDataString .text 00414450 00000073 R . . . B . .
_RTC_UninitUse .text 004144E0 0000010C R . . . B . .
_RTC_NumErrors .text 00414630 00000006 R . . . . .
_RTC_GetErrDesc .text 00414640 00000018 R . . . B . .
_RTC_SetErrorType .text 00414660 00000023 R . . . B . .
_RTC_SetErrorFunc .text 00414690 0000001D R . . . B . .
_RTC_SetErrorFuncW .text 004146C0 0000001D R . . . B . .
_RTC_GetErrorFunc .text 004146F0 00000006 R . . . . .
_RTC_GetErrorFuncW .text 00414700 00000006 R . . . . .
__CRT_RTC_INITW .text 00414706 00000006 R . . . . .
__report_gsfailure .text 00414710 0000010C R . . . B . .
__configthreadlocale .text 00414860 00000006 R . . . . .
__setdefaultprecision .text 00414870 00000035 R . . . B . .
__invoke_watson_if_error .text 004148C0 00000029 R . . . B . .
__setusermatherr .text 004148F4 00000006 R . . . . .
__matherr .text 00414900 00000007 R . . . B . .
__setargv .text 00414910 00000007 R . . . B . .
_RTC_Initialize .text 00414920 00000025 R . . . . .
_RTC_Terminate .text 00414950 00000025 R . . . . .
__p_commode .text 0041497E 00000006 R . . . . .
__p_fmode .text 00414984 00000006 R . . . . .
__onexit .text 00414990 000000D9 R . . . B T .
sub_414A69 .text 00414A69 0000000B R . . . . .
atexit .text 00414AD0 0000001A R . . . B T .
__encode_pointer .text 00414AF0 00000006 R . . . . .
__set_app_type .text 00414AF6 00000006 R . . . . .
__amsg_exit .text 00414AFC 00000006 R . . . . .
__getmainargs .text 00414B02 00000006 R . . . . .
__security_init_cookie .text 00414B10 0000011A R . . . B . .
__exit .text 00414C70 00000006 . . . . T .
__XcptFilter .text 00414C76 00000006 R . . . . .
__cexit .text 00414C7C 00000006 R . . . . T .
__CrtSetCheckCount .text 00414C82 00000006 R . . . . .
_ValidateImageBase .text 00414C90 0000005D R . . . B . .
_FindPESection .text 00414D10 00000073 R . . . B . .
_IsNonwritableInCurrentImage .text 00414DA0 00000110 R . . . B . .
__CrtDbgReportW .text 00414EF4 00000006 R . . . . .
__initterm .text 00414EFA 00000006 R . . . . .
__initterm_e .text 00414F00 00000006 R . . . . .
__except_handler4 .text 00414F10 00000027 R . . . B . .
_RTC_GetSrcLine .text 00414F40 000002C2 R . . . B T .

```

GetPdbDll	.text 004152C0 000001BC R . . . B . .
__crt_debugger_hook	.text 004154EC 00000006 R . . . . .
__controlfp_s	.text 004154F2 00000006 R . . . . .
__invoke_watson	.text 004154F8 00000006 R . . . . .
__unlock	.text 004154FE 00000006 R . . . . .
__dllonexit	.text 00415504 00000006 R . . . . .
__lock	.text 0041550A 00000006 R . . . . .
__decode_pointer	.text 00415510 00000006 R . . . . .
__except_handler4_common	.text 00415516 00000006 R . . . . .
InterlockedExchange(x,x)	.text 0041551C 00000006 R . . . . T .
Sleep(x)	.text 00415522 00000006 R . . . . T .
InterlockedCompareExchange(x,x,x)	.text 00415528 00000006 R . . . . T .
RaiseException(x,x,x,x)	.text 0041552E 00000006 R . . . . T .
DebugBreak()	.text 00415534 00000006 R . . . . T .
WideCharToMultiByte(x,x,x,x,x,x,x,x)	.text 0041553A 00000006 R . . . . T .
IsDebuggerPresent()	.text 00415540 00000006 R . . . . T .
MultiByteToWideChar(x,x,x,x,x,x)	.text 00415546 00000006 R . . . . T .
IstrlenA(x)	.text 0041554C 00000006 R . . . . T .
GetProcAddress(x,x)	.text 00415552 00000006 R . . . . T .
LoadLibraryA(x)	.text 00415558 00000006 R . . . . T .
TerminateProcess(x,x)	.text 0041555E 00000006 R . . . . T .
GetCurrentProcess()	.text 00415564 00000006 R . . . . T .
UnhandledExceptionFilter(x)	.text 0041556A 00000006 R . . . . T .
SetUnhandledExceptionFilter(x)	.text 00415570 00000006 R . . . . T .
QueryPerformanceCounter(x)	.text 00415576 00000006 R . . . . T .
GetTickCount()	.text 0041557C 00000006 R . . . . T .
GetCurrentThreadId()	.text 00415582 00000006 R . . . . T .
GetCurrentProcessId()	.text 00415588 00000006 R . . . . T .
GetSystemTimeAsFileTime(x)	.text 0041558E 00000006 R . . . . T .
FatalAppExitA(x,x)	.text 00415594 00000006 R . . . . T .
HeapFree(x,x,x)	.text 0041559A 00000006 R . . . . T .
HeapAlloc(x,x,x)	.text 004155A0 00000006 R . . . . T .
GetProcessHeap()	.text 004155A6 00000006 R . . . . T .
GetModuleFileNameW(x,x,x)	.text 004155AC 00000006 R . . . . T .
VirtualQuery(x,x,x)	.text 004155B2 00000006 R . . . . T .
FreeLibrary(x)	.text 004155B8 00000006 R . . . . T .

Appendix Figure B: IDApro function dump for Figures 6-13. Hit funcs are in bold.

Type:ASCII Size:11 Data:calculate  
Type:ASCII Size:6 Data:cmd  
Type:ASCII Size:3 Data:0  
Type:ASCII Size:3 Data:2  
Type:ASCII Size:3 Data:3  
Type:ASCII Size:3 Data:4  
Type:ASCII Size:3 Data:5  
Type:ASCII Size:3 Data:6  
Type:ASCII Size:3 Data:7  
Type:ASCII Size:3 Data:8  
Type:ASCII Size:3 Data:9

Appendix Figure C: The *TextServer* seedfile