Profiling

- How do you know what your program does?
  - How much time does your program spend in which function?
  - How often are specific functions called?
  - What can this tell us?
    - Which functions take more/less time than you expected?
    - Which functions get called more/less than you expected?

Profiling

- Profiling requires
  - compiling and linking the program with profiling enabled
  - running the program to generate profiling data
  - running a profiler (e.g., *gprof*) to analyze the profiling data

Profiling

- Profiling can do more than just
  - see how much time we spend where
  - how often a function is called
  - etc.
- Profiling can be used to detect ongoing attacks
  - let’s take a look at an attack as it unfolds in time
  - the demo is from an attack called *hiperbomb2*
- Let’s look at the latter first...

Profiles

- We view a system as a collection of profiles of its functionalities P_i

$$P_{sys}(\Delta t) = \sum_{i=1}^{k} P_i(\Delta t)$$

k is the number of functionalities active during Δt

- Functionality Profile

$$P_i(\Delta t) = (f_1(\Delta t), f_2(\Delta t), \ldots, f_n(\Delta t))$$

$f_j(\Delta t)$ is the number of times identity F_j has been invoked during Δt

Attack Signatures

- Atomic Attacks A_i
  - the smallest attack technology unit
  - e.g. a port sweep, sequence of unsuccessful login attempts
- Attack Signature S_i
  - the portion of a profile that is attributable to A_i

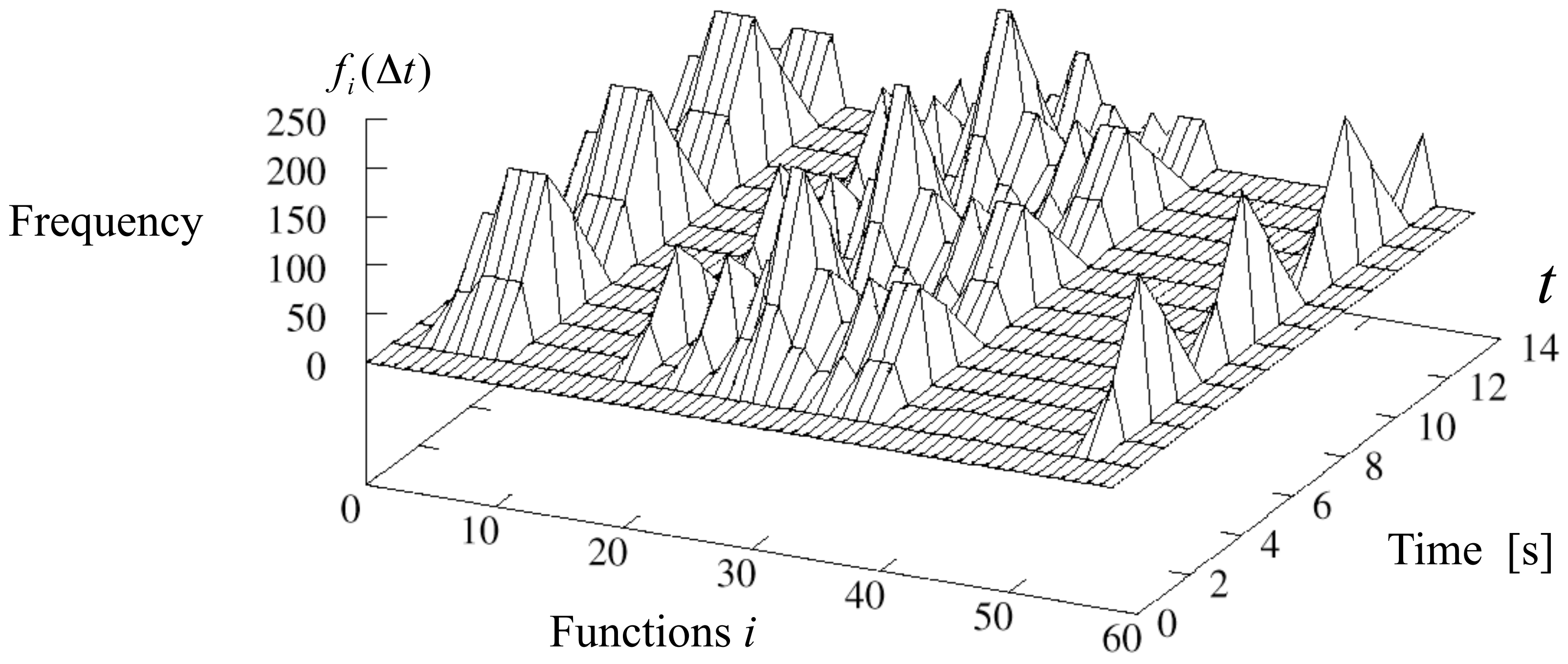
$$S_i(\Delta t) = (f_{\alpha(1)}(\Delta t), f_{\alpha(2)}(\Delta t), \ldots, f_{\alpha(s_i)}(\Delta t))$$

α is a one-to-one mapping from indices of S_i to indices of the identities F_j profiled

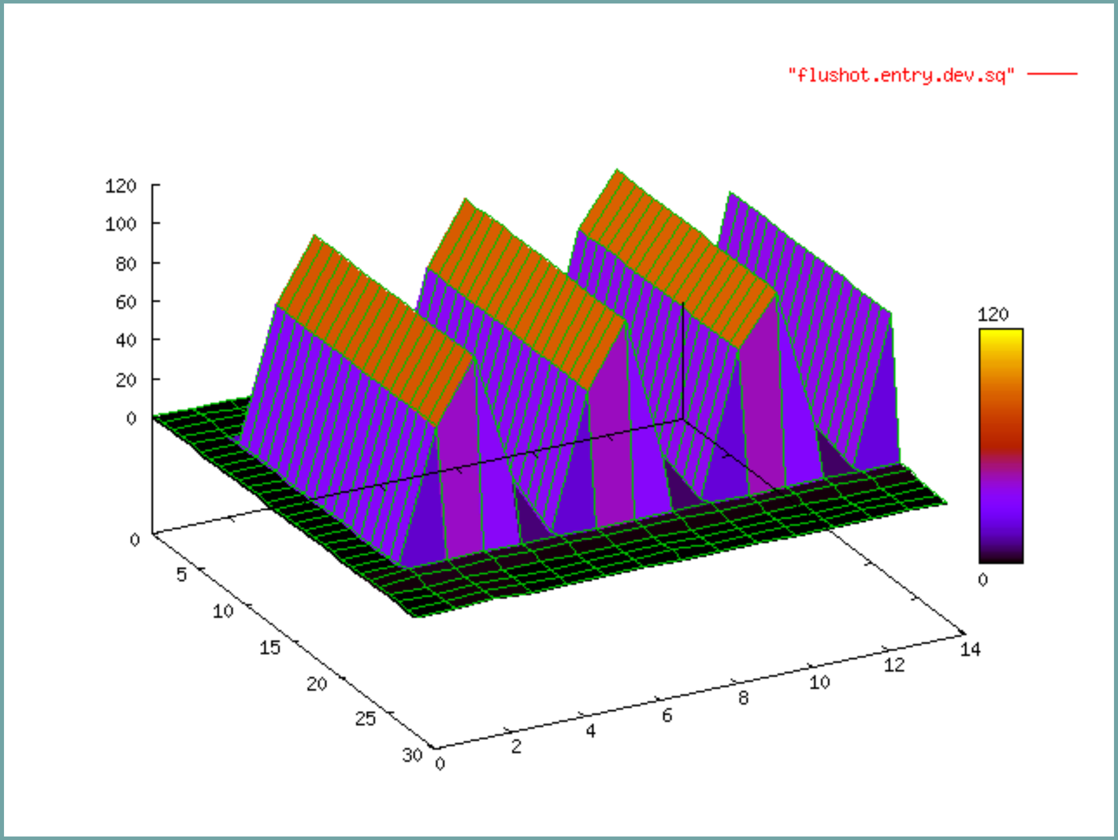
Attack Signature

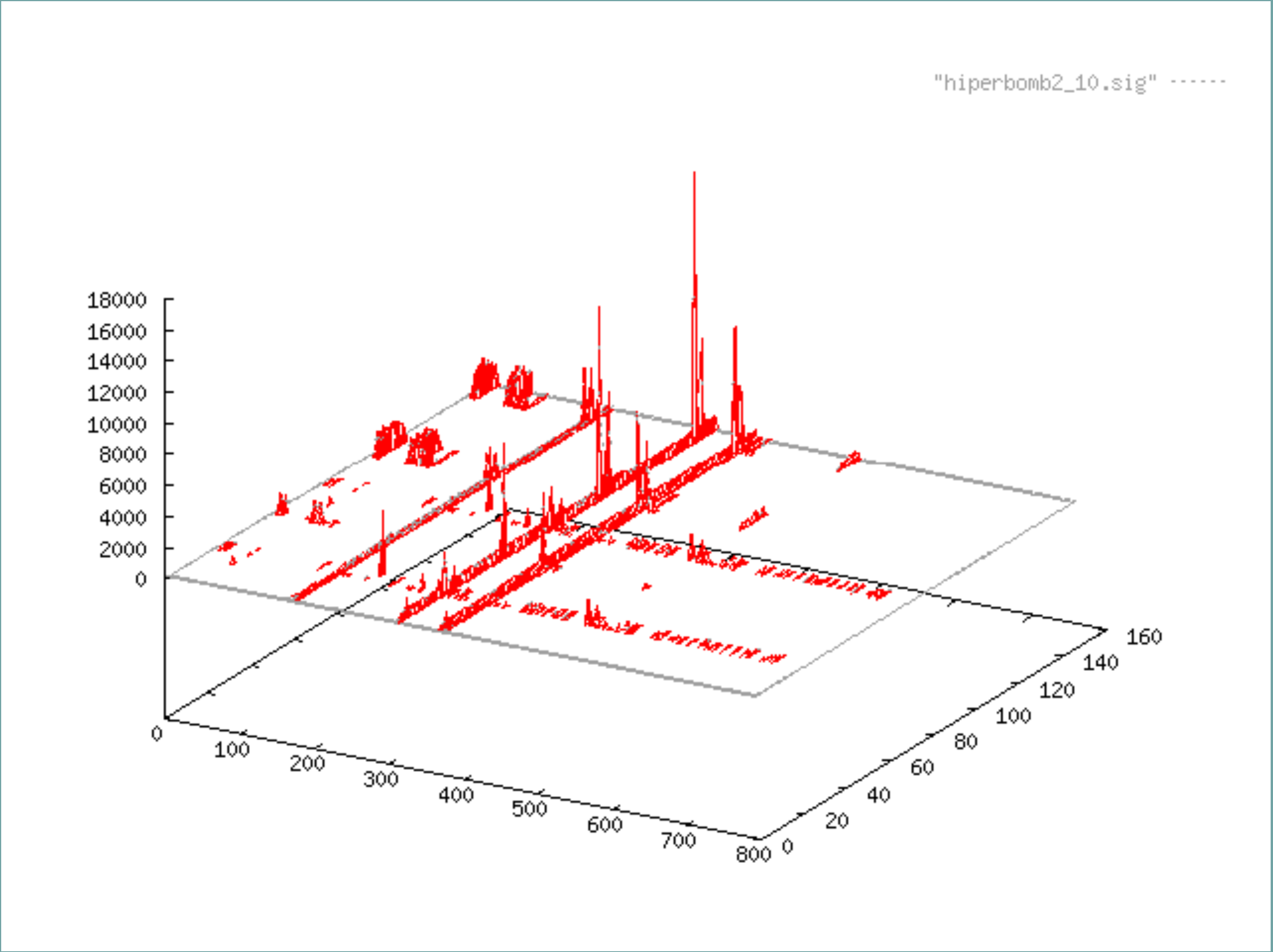
- Attack Signature over Time
  - Example: “teardrop”

    (overlapping IP(TCP) fragments are formatted to cause reassembly crashes)



A Three-Dimensional Profile



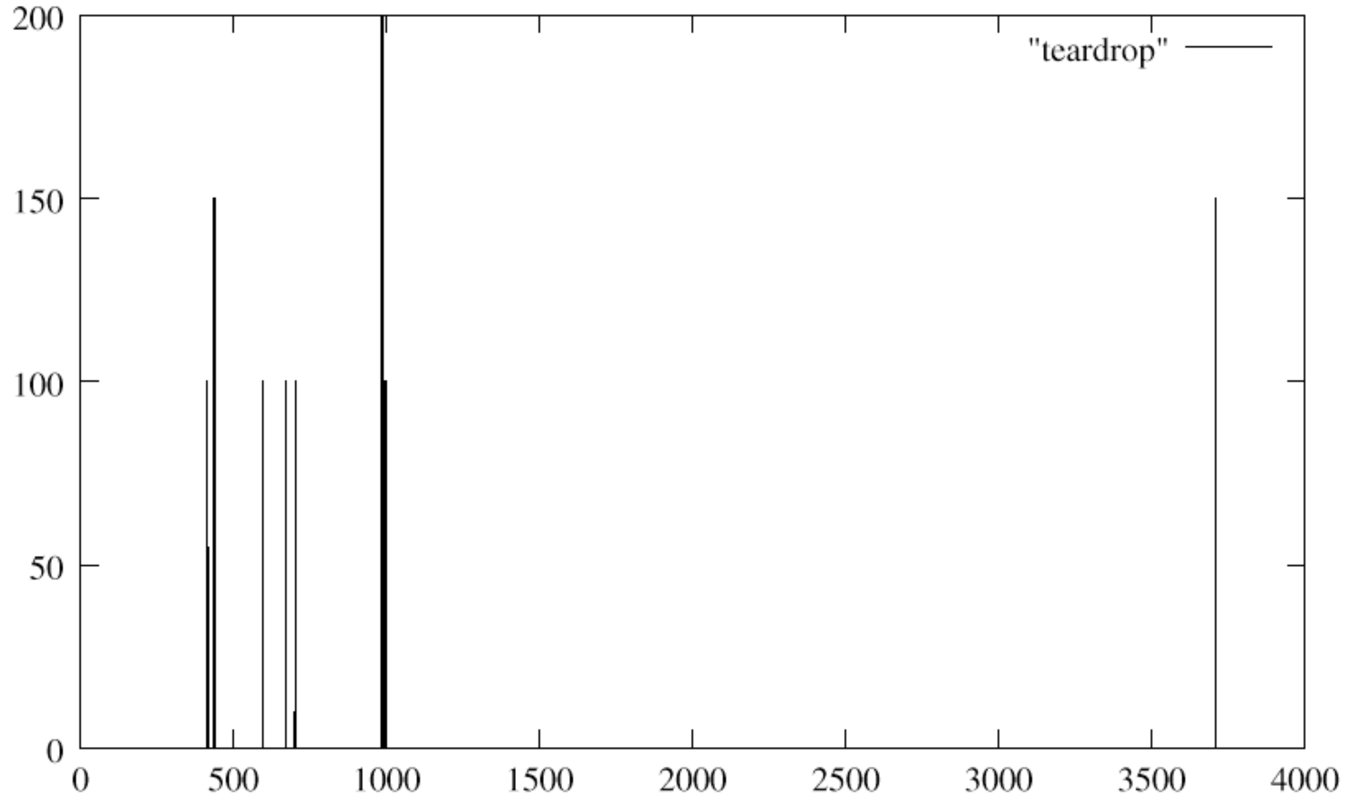

"hiperbomb2_10.sig"
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0
100
200
300
400
500
600
700
800
0
20
40
60
80
100
120
140
160

Attack Signature

- Example “teardrop”

Frequency

$f_i(\Delta t)$



Functions i

Real-Time Attack Recognition

- Vector Analysis
  - Profile $P_i(\Delta t)$, Idle Signature $S_0(\Delta t)$, and Attack Signature $S_i(\Delta t)$ are vectors
- "Strictly Speaking"
  - there are three possible scenarios

$$P_{sys}(\Delta t) \geq S_i(\Delta t) \ \text{possible attack}$$

$$P_{sys}(\Delta t) \neq S_i(\Delta t) \ \text{attack not possible}$$

$$P_{sys}(\Delta t) < S_i(\Delta t) \ \text{attack not possible}$$

Signature Analysis

- Relationship between Signatures

$$\mathbf{S}_i \subseteq \mathbf{S}_j$$

- Common functions

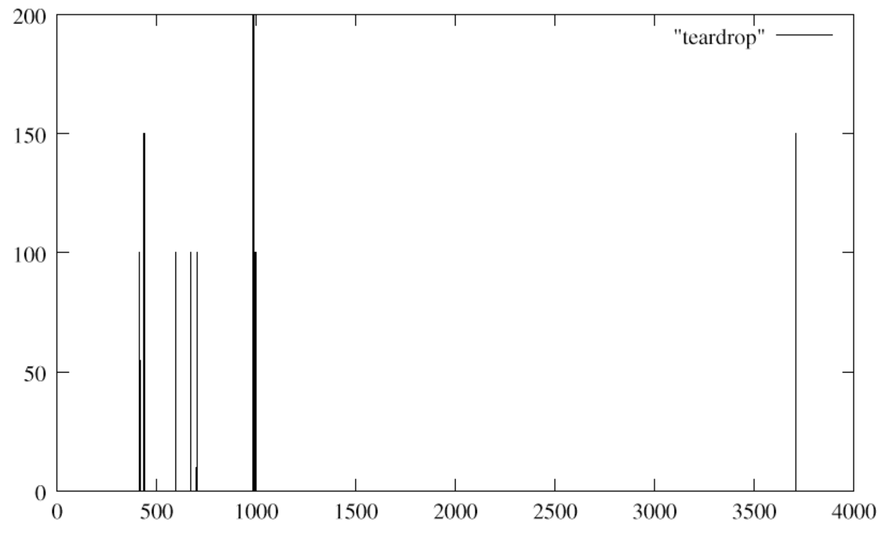
$$\mathbf{S}_i \cap \mathbf{S}_j$$

- Signature Correlation

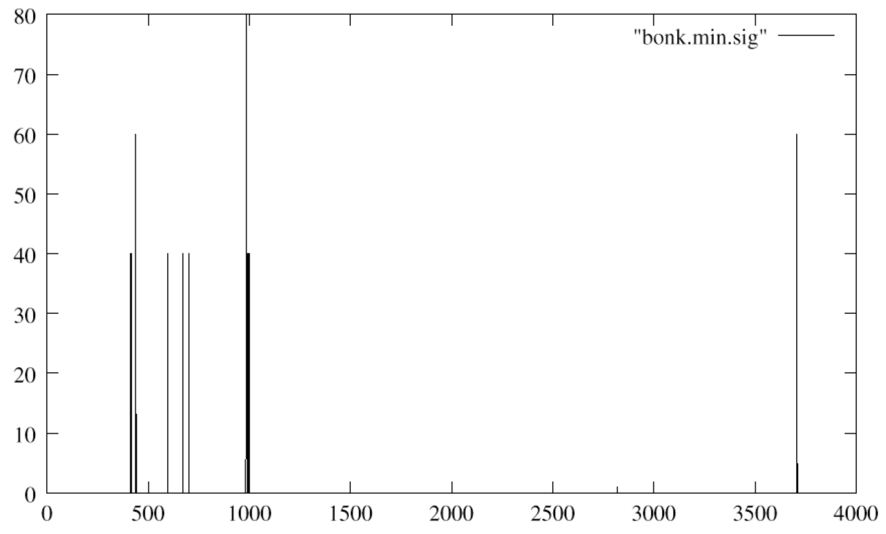
$$C(i,j) = \frac{|\mathbf{S}_i \cap \mathbf{S}_j|}{\min(|\mathbf{S}_i|, |\mathbf{S}_j|)}$$

Attack Signature

- Example “teardrop” vs. “bonk”
  - bonk: malformed IP header causes packet size violation upon reassembly
  - Note: scales differ
  - Correlation is 1.0



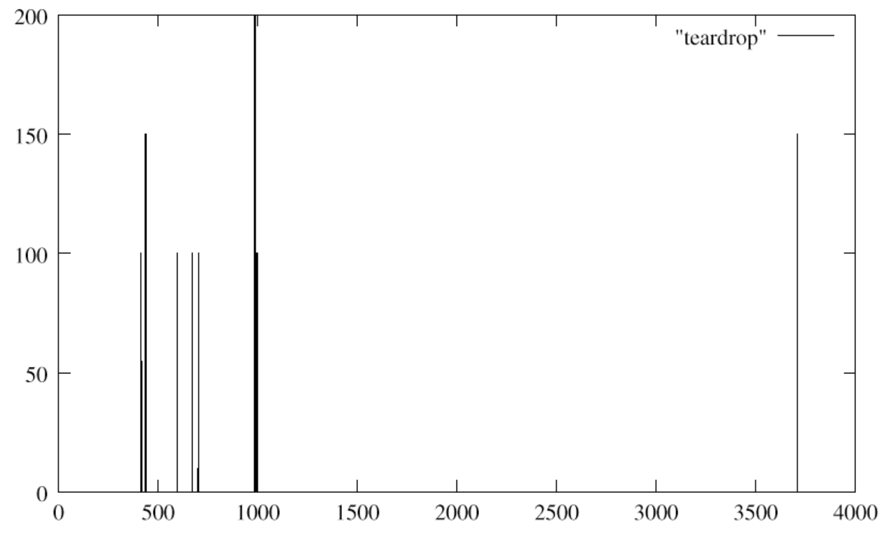
teardrop attack



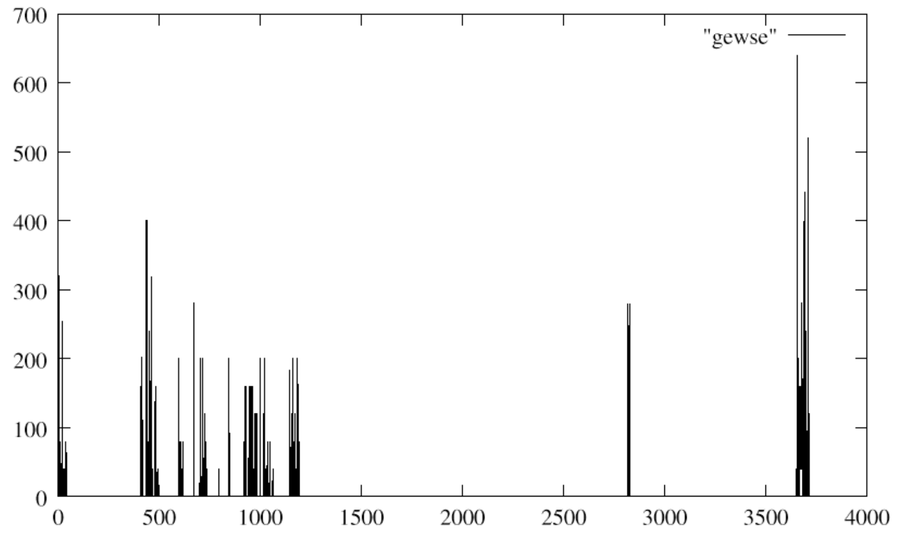
bonk attack

Attack Signature

- Example "teardrop" vs. "gewse"
  - Gewse: (DoS - attack) floods identd on port 139
  - Note: scales differ
  - Correlation is 0.54



teardrop attack



gewse attack

Correlation

- “Some things seem too good to be true”

		13	13	18	18	18	18	20	20	21	21	21	21	21	22	22	24	35	35	37	42	45	51	57	135	164	194
		conseal	misfrag	fawx	jolt	pimp2	ssping	flushot	trash	boink	bonk	newtear	syndrop	teardrop	nestea	smack	dcd3c	beer	spiffit	biffit	synhose	land	pepsi	trash2	gewse	gewse5	hiperbomb2
13	conseal	1.00	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.92	0.92	0.92	0.85	0.92	0.92	0.85	0.85	0.85
13	misfrag	0.85	1.00	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.85	0.92	0.92	1.00	0.85	0.85	1.00	0.77	0.85	0.85	1.00	1.00	1.00
18	faw x	0.85	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.61	0.61	0.61	0.61	0.61	0.61	0.67	0.61	1.00	0.61	0.61	0.61
18	jolt	0.85	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.61	0.61	0.61	0.61	0.61	0.61	0.67	0.61	1.00	0.61	0.61	0.61
18	pimp2	0.85	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.61	0.61	0.61	0.61	0.61	0.61	0.67	0.61	1.00	0.61	0.61	0.61
18	ssping	0.85	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.61	0.61	0.61	0.61	0.61	0.61	0.67	0.61	1.00	0.61	0.61	0.61
20	flushot	0.85	0.85	1.00	1.00	1.00	1.00	1.00	1.00	0.90	0.90	0.90	0.90	0.90	0.90	0.55	0.55	0.55	0.55	0.55	0.65	0.60	0.55	1.00	0.65	0.60	0.55
20	trash	0.85	0.85	1.00	1.00	1.00	1.00	1.00	1.00	0.90	0.90	0.90	0.90	0.90	0.90	0.55	0.55	0.55	0.55	0.55	0.65	0.60	0.55	1.00	0.65	0.60	0.55
21	boink	0.85	0.85	1.00	1.00	1.00	1.00	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.52	0.52	0.52	0.52	0.52	0.52	0.62	0.52	1.00	0.52	0.52	0.52
21	bonk	0.85	0.85	1.00	1.00	1.00	1.00	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.52	0.52	0.52	0.52	0.52	0.52	0.62	0.52	1.00	0.52	0.52	0.52
21	new tear	0.85	0.85	1.00	1.00	1.00	1.00	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.52	0.52	0.52	0.52	0.52	0.52	0.62	0.52	1.00	0.52	0.52	0.52
21	syndrop	0.85	0.85	1.00	1.00	1.00	1.00	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.52	0.52	0.52	0.52	0.52	0.52	0.62	0.52	1.00	0.52	0.52	0.52
21	teardrop	0.85	0.85	1.00	1.00	1.00	1.00	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.52	0.52	0.52	0.52	0.52	0.52	0.62	0.52	1.00	0.52	0.52	0.52
22	nestea	0.85	0.85	1.00	1.00	1.00	1.00	0.90	0.90	1.00	1.00	1.00	1.00	1.00	1.00	0.50	0.50	0.50	0.50	0.50	0.50	0.59	0.50	0.95	0.50	0.50	0.50
22	smack	0.85	0.92	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	1.00	1.00	0.55	0.59	0.59	0.73	0.64	0.77	0.68	0.55	0.55	0.55
24	dcd3c	0.85	0.92	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	1.00	1.00	0.50	0.54	0.54	0.75	0.63	0.75	0.67	0.50	0.50	0.50
35	beer	0.85	1.00	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	0.55	0.50	1.00	0.77	0.77	0.57	0.71	0.77	0.77	0.80	0.74	0.80
35	spiffit	0.92	0.85	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	0.59	0.54	0.77	1.00	1.00	0.43	0.86	1.00	0.91	0.66	0.60	0.66
37	biffit	0.92	0.85	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	0.59	0.54	0.77	1.00	1.00	0.41	0.86	1.00	0.92	0.62	0.57	0.62
42	synhose	0.92	1.00	0.61	0.61	0.61	0.61	0.65	0.65	0.52	0.52	0.52	0.52	0.52	0.50	0.73	0.75	0.57	0.43	0.41	1.00	0.50	0.50	0.57	0.67	0.64	0.62
45	land	0.85	0.77	0.67	0.67	0.67	0.67	0.60	0.60	0.62	0.62	0.62	0.62	0.62	0.59	0.64	0.63	0.71	0.86	0.86	0.50	1.00	0.87	0.96	0.47	0.42	0.47
51	pepsi	0.92	0.85	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	0.77	0.75	0.77	1.00	1.00	0.50	0.87	1.00	0.86	0.45	0.41	0.45
57	trash2	0.92	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.95	0.68	0.67	0.77	0.91	0.92	0.57	0.96	0.86	1.00	0.44	0.39	0.40
135	gew se	0.85	1.00	0.61	0.61	0.61	0.61	0.65	0.65	0.52	0.52	0.52	0.52	0.52	0.50	0.55	0.50	0.80	0.66	0.62	0.67	0.47	0.45	0.44	1.00	0.99	0.95
164	gew se5	0.85	1.00	0.61	0.61	0.61	0.61	0.60	0.60	0.52	0.52	0.52	0.52	0.52	0.50	0.55	0.50	0.74	0.60	0.57	0.64	0.42	0.41	0.39	0.99	1.00	0.96
194	hiperbomb2	0.85	1.00	0.61	0.61	0.61	0.61	0.55	0.55	0.52	0.52	0.52	0.52	0.52	0.50	0.55	0.50	0.80	0.66	0.62	0.62	0.47	0.45	0.40	0.95	0.96	1.00

Profiling

- GNU Profiler: gprof
  - Utility: *gprof* -b [*executableFile* [*profileFile*]]
    - *gprof* generates a table of time and repetitions of each function in the executable *executableFile* based on the performance trace stored in the file *profileFile*. If profileFile or executableFile are omitted, "gmon.out" or "a.out" is assumed respectively.
    - The executable file must have been compiled using the **-pg** option of gcc, which instructs the compiler to generate special code that writes a "gmon.out" file when the program runs.
    - The gprof utility looks at this output file after the program has terminated and displays the information. The output of gprof is verbose (but helpful); to instruct gprof to be brief, use the **-b** option.

Profiling

- For more information on GNU gprof check out
  - http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html#SEC1
  - the rest of the profiling discussion presented here is based on their discussion and the examples are restated
  - note that the authors are using cc rather than gcc. Check your Linux system and you will likely see a link from cc to gcc

Profiling

- Execution to generate profiling data
  - Compilation must specify the **-pg** option
    - this option works with compilation and linking
  - Deterministic vs nondeterministic execution
    - does you program depend on the value of arguments?
    - how about other dependencies, e.g., time, file size, number of users etc. -- all of that may or will have changed the next time you run the program
  - Program must exit normally for the file *gmon.out* to be generated

Profiling

- Flat Profile
  - shows the total number of time spent in each function
  - unless explicitly indicated (**-z** option) zero time functions are not listed
  - a function not compiled with **-pg** is indistinguishable from a function that was never called

Profiling

- example from above cited source

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
```

Profiling

- Interpretation of example
  - functions *mcount* and *profile* are part of profiling and their time represents pure profiling overhead
  - columns
    - *% time*: total execution time of program spent in this function
    - *cumulative seconds*: time spent in the function and everything above it in the table
    - *self seconds*: time spent in the function alone, which is the time that determines the position of the function in the list
    - *calls*: the total number of times the function was called. A function that was never called or was not compiled for profiling will show a blank field here.

Profiling

- Interpretation of example
  - columns, cont.
    - *self ms/call*: the average number of milliseconds spent in the function per call
    - *total ms/call*: average number of ms spent in this function and its dependents per call
    - *name*: the name of the function

Profiling

- Call Graph
  - A dependency graph reflecting the caller callee relationship
  - Static call graph
    - shows all dependancies the program implies
  - Dynamic call graph
    - the call graph as it unfolds during execution

Profiling

- gprof call graph
  - shows ho much time was spent in each function and its children
  - can use to find functions that may not use much time, but that call functions that use much time.

Profiling

- example from above cited source

```
granularity: each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1         main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1         report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
```

Profiling

- Interpretation of example
  - dashed lines divide table into entries
    - one entry for each function
    - entry many have one or more lines
  - primary line is indicate by number in [] and shows associated function name
    - preceding lines of entry describe callers (i.e., parents)
    - succeeding lines describe its subroutines (i.e., children)
  - entries are sorted by how much time is spent in the function <u>and</u> its subroutines (i.e., children)

Profiling

- Primary line
  - e.g.: index % time self children called name
  - e.g.: [3] 100.0 0.00 0.05 1 report [3]
- columns
  - *index*: index number of the consecutively numbered function
  - *% time*: fraction of total time spent in this function, including time spent in its children
  - *self*: amount of time spent by the function
  - *children*: total amount of time spent in its children
  - *called*: number of times the function was called
  - *name*: name of the current function (with index repeated)

Profiling

- cycles, e.g.

```
[4]   59.8   0.01   0.02      8+472  <cycle 2 as a whole>    [4]
             0.01   0.02   244+260       offtime <cycle 2> [7]
             0.00   0.00   236+1         tzset <cycle 2> [26]
```

  - columns
    - *name*: If function is part of cycle of recursion, the cycle number is printed between the function’s name and the index number
    - e.g. function offtime is part of <cycle 2>

Profiling

- Lines for a Function’s Caller
  - function’s entry has a line for each function it was called by

```
index  % time    self  children called     name
...
                 0.00    0.05       1/1           main [2]
[3]    100.0     0.00    0.05       1           report [3]
```

  - *self*: estimate on time spent in function *report* itself when it was called from *main*
  - *children*: estimate of time spent in children of *report* when it was called from *main*
  - *called*: x/y, x=#of times *report* was called from *main,* y= total number of non-recursive calls to *report* from all its callers

Profiling

- Lines for a Function's Subroutines (children)
  - function's entry has a line for each of its subroutines

```
index   % time     self   children called      name
...
[2]     100.0     0.00     0.05        1           main [2]
                  0.00     0.05        1/1             report [3]
```

  - *self*: estimate of time spent directly within *report* when report was called from *main*.
  - *children*: estimate of time spent in children of *report* when it was called from *main*
  - *called*: x/y, x=#of calls to *report* from *main,* y= total number of non-recursive calls to *report*

Profiling

- Statistical inaccuracy of grpof output
  - gprof samples run-times => subject to statistical inaccuracy
    - sampling period is given at top of flat profile
    - e.g. *Each sample counts as 0.01 seconds*.
    - run-time info is accurate if it is considerably larger than the sampling period. Why is that?
  - Number of calls are derived by counting, not sampling

Profiling

- Statistical inaccuracy of grpof output
  - Get more accuracy by running program longer; use **-s** option of *grpof*

1. Run your program once.
2. Issue the command `` `mv gmon.out gmon.sum' ``.
3. Run your program again, the same as before.
4. Merge the new data in `` `gmon.out' `` into `` `gmon.sum' `` with this command:
   ```
   gprof -s executable-file gmon.out gmon.sum
   ```
5. Repeat the last two steps as often as you wish.
6. Analyze the cumulative data using this command:
   ```
   gprof executable-file gmon.sum > output-file
   ```

Debugging

- The GNU debugger **gdb** allows to symbolically debug a program. You can
  - run and list the program
  - set breakpoints
  - examine variable values
  - trace execution

Debugging

- Utility: **gdb** *executableFilename*
  - **gdb** is a standard GNU/Linux debugger.
  - The named executable file is loaded into the debugger and a user prompt is displayed.
  - To obtain information on the various **gdb** commands, enter **help** at the prompt.

- Read the debugging section of the book and play with the debugger!
  - This is something you need to do at your own pace.

strip

- What does the debugger or profiler add to the code?
  - Extra code to do the things it does
  - This is pure overhead
  - One can strip this code with **strip**
  - Synopsis: **strip** { *fileName* }+
    - **strip** removes all of the symbol table, relocation, debugging, and profiling information from the named files.