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 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 $\Delta t$

- 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 $\Delta t$
**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 $A_i$

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

$\alpha$ 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

What does it look like?
**Attack Signature**

- Example “teardrop”

![frequency Vs. functions.png](image)

**Real-Time Attack Recognition**

- Vector Analysis
  - Profile $P_i(\Delta t)$, Idle Signature $S_i(\Delta t)$, and Attack Signature $S_i(\Delta t)$ are vectors

- “Strictly Speaking”
  - there are three possible scenarios

  
  \[
  P_{sys}(\Delta t) \geq S_i(\Delta t) \quad \text{possible attack}
  \]

  
  \[
  P_{sys}(\Delta t) \nless S_i(\Delta t) \quad \text{attack not possible}
  \]

  
  \[
  P_{sys}(\Delta t) < S_i(\Delta t) \quad \text{attack not possible}
  \]
**Signature Analysis**

- Relationship between Signatures
  \[ S_i \subseteq S_j \]
- Common functions
  \[ S_i \cap S_j \]
- Signature Correlation
  \[ C(i, j) = \frac{|S_i \cap S_j|}{\min(|S_i|, |S_j|)} \]

**Attack Signature**

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

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

**Correlation**

- “Some things seem too good to be true”

---

### Correlation Table

<table>
<thead>
<tr>
<th>Example 1</th>
<th>Example 2</th>
<th>Example 3</th>
<th>Example 4</th>
<th>Example 5</th>
<th>Example 6</th>
<th>Example 7</th>
<th>Example 8</th>
<th>Example 9</th>
<th>Example 10</th>
<th>Example 11</th>
<th>Example 12</th>
<th>Example 13</th>
<th>Example 14</th>
<th>Example 15</th>
<th>Example 16</th>
<th>Example 17</th>
<th>Example 18</th>
<th>Example 19</th>
<th>Example 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>teardrop</td>
<td>0.85</td>
<td>0.86</td>
<td>0.87</td>
<td>0.88</td>
<td>0.89</td>
<td>0.90</td>
<td>0.91</td>
<td>0.92</td>
<td>0.93</td>
<td>0.94</td>
<td>0.95</td>
<td>0.96</td>
<td>0.97</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>gewse</td>
<td>0.90</td>
<td>0.91</td>
<td>0.92</td>
<td>0.93</td>
<td>0.94</td>
<td>0.95</td>
<td>0.96</td>
<td>0.97</td>
<td>0.98</td>
<td>0.99</td>
<td>0.96</td>
<td>0.97</td>
<td>0.98</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>
Profiling

- GNU Profiler: gprof


  - `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.

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 your 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.

<table>
<thead>
<tr>
<th>% cumulative time</th>
<th>cumulative seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.34</td>
<td>0.02</td>
<td>0.02</td>
<td>7208</td>
<td>0.00</td>
<td>0.00</td>
<td>open</td>
</tr>
<tr>
<td>16.67</td>
<td>0.03</td>
<td>0.01</td>
<td>244</td>
<td>0.04</td>
<td>0.12</td>
<td>offtime</td>
</tr>
<tr>
<td>16.67</td>
<td>0.04</td>
<td>0.01</td>
<td>8</td>
<td>1.25</td>
<td>1.25</td>
<td>memccpy</td>
</tr>
<tr>
<td>16.67</td>
<td>0.05</td>
<td>0.01</td>
<td>7</td>
<td>1.43</td>
<td>1.43</td>
<td>write</td>
</tr>
<tr>
<td>16.67</td>
<td>0.06</td>
<td>0.01</td>
<td>mcount</td>
<td>0.00</td>
<td>0.00</td>
<td>mcount</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>236</td>
<td>0.00</td>
<td>0.00</td>
<td>tzset</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>192</td>
<td>0.00</td>
<td>0.00</td>
<td>tolower</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>47</td>
<td>0.00</td>
<td>0.00</td>
<td>strlen</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>45</td>
<td>0.00</td>
<td>0.00</td>
<td>strchr</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>main</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>memccpy</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>print</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>profil</td>
</tr>
<tr>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>50.00</td>
<td>report</td>
</tr>
</tbody>
</table>

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 how 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

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>spontaneous</td>
</tr>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td></td>
<td>start [1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.05</td>
<td>1/1</td>
<td>main [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/2</td>
<td>on_exit [28]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td>exit [59]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;cycle 1&gt;</td>
</tr>
<tr>
<td>[2]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>1</td>
<td>main [2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.05</td>
<td>1/1</td>
<td>report [3]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;cycle 2 as a whole&gt; [4]</td>
</tr>
<tr>
<td>[3]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>1</td>
<td>report [3]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.03</td>
<td>8/8</td>
<td>timelocal [6]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.01</td>
<td>1/1</td>
<td>print [9]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.01</td>
<td>9/9</td>
<td>fgets [12]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>12/34</td>
<td>strcmp &lt;cycle 1&gt; [40]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>8/8</td>
<td>lookup [20]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>1/1</td>
<td>fopen [21]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>8/8</td>
<td>chewtime [24]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>8/16</td>
<td>skipspace [44]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&lt;cycle 2&gt;</td>
</tr>
<tr>
<td>[4]</td>
<td>59.8</td>
<td>0.01</td>
<td>0.02</td>
<td>8+472</td>
<td>&lt;cycle 2 as a whole&gt; [4]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.01</td>
<td>0.02</td>
<td>244+260</td>
<td>offtime &lt;cycle 2&gt; [7]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>236+1</td>
<td>tzset &lt;cycle 2&gt; [26]</td>
</tr>
</tbody>
</table>

---

Granularity: Each sample hit covers 2 byte(s) for 20.00% of 0.05 seconds
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 indicated 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 and its subroutines (i.e., children)

- 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.

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>1/1 main</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>1 report</td>
</tr>
</tbody>
</table>

- *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`, $y=\text{total number of non-recursive calls to } report\text{ from all its callers}$
Profiling

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

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100.0</td>
<td>0.00</td>
<td>0.05</td>
<td>main [2]</td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.05</td>
<td>1/1</td>
<td>report [3]</td>
</tr>
</tbody>
</table>

- *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
  - grpof 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.