

# Stringer: Measuring the Importance of Static Data Comparisons to Detect Backdoors and Undocumented Functionality

**Sam L. Thomas**, Tom Chothia, Flavio D. Garcia

School of Computer Science  
University of Birmingham  
Birmingham  
United Kingdom  
B15 2TT

`{s.l.thomas,t.p.chothia,f.garcia}@cs.bham.ac.uk`

European Symposium on Research in Computer Security (ESORICS) 2017

# Challenge

- How do we reduce the manual effort required to identify undocumented functionality and backdoors within software?

# Challenge

- How do we reduce the manual effort required to identify undocumented functionality and backdoors within software?



- Undocumented functionality? Backdoors?
  - Authentication bypass by “magic” words.
  - Hard-coded credential checks.
  - Additional protocol messages that activate unexpected functionality.

# Application

Focus on embedded device firmware – it's a challenging target:

- Lots of devices, lots of firmware.
- Multiple firmware versions for each device.
- Impossible to manually analyse *every* firmware image.

Stringer

# Objective

- Identify interesting code structures and static data comparisons that lead to backdoor-like behaviour.
- Lightweight analysis.

- ① Automatically identify static data comparison functions.
- ② A metric for measuring the degree a binary's functions branching is influenced by comparisons with static data.



For a given binary:

- 1 Identify all possible static data comparison functions:

`strcmp`

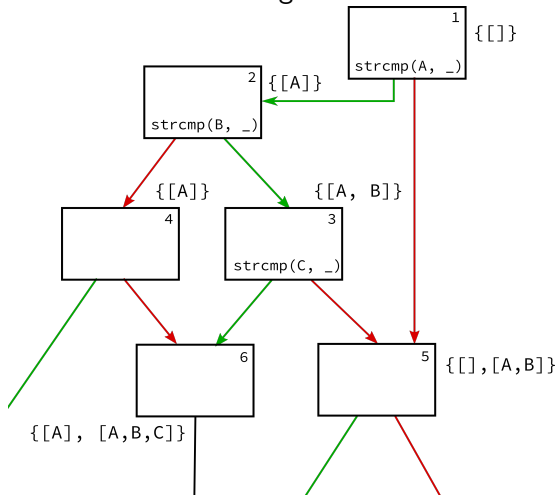
`memcmp`

`strstr`

`strncmp`

`strcasecmp`

- 2 Label the basic blocks of all functions with the sets of static data sequences that must be matched against to reach them:



- Using the computed sets, calculate a score for each element of static data:

$$A = 100$$

$$B = 200$$

...

- Using the computed sets, calculate a score for each element of static data:

$$A = 100$$

$$B = 200$$

...

- Finally, using the scores for each item of static data, compute a score for each function:

$$f = 300$$

...

# Identifying Static Data Comparison Functions

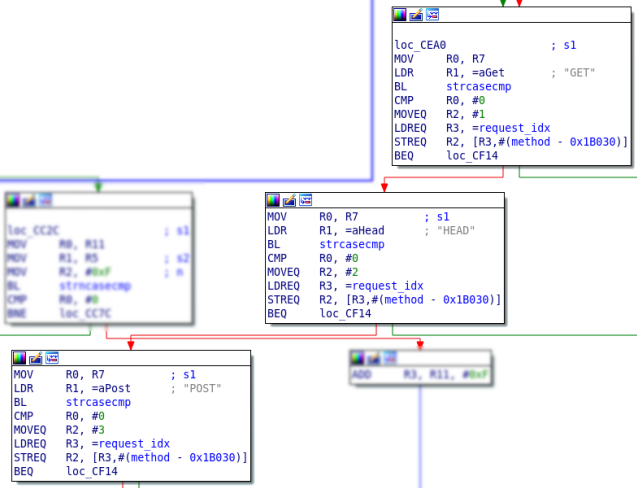
# Identifying static data comparison functions

Approach based upon concrete observations:

- Analyse calls to static data comparison functions in C/C++ binaries.
- Collect properties that are common amongst them: call-sites, number of arguments, how they influence branching, ...

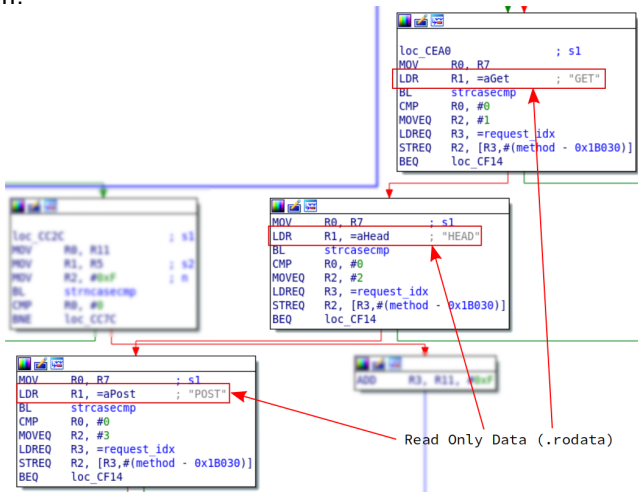
# Motivating Example

HTTP protocol parser from mini\_httpd binary:



# Call-site Properties

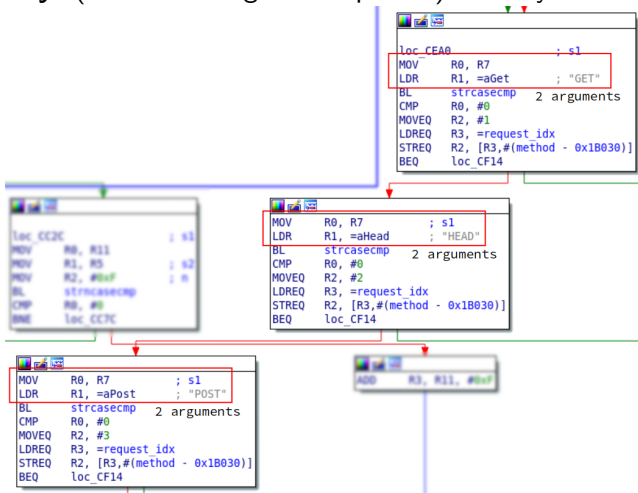
**Argument references:** at least one argument refers to the data/read-only data section:





# Call-site Properties

Function arity: (number of arguments passed): usually 2-3:



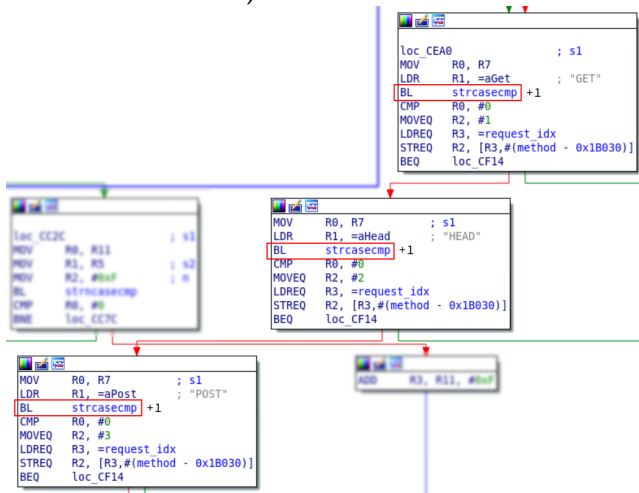
# Call-site Properties

Branching properties: boolean comparison (i.e. matches or not):



# Call-site Properties

**Local call frequency:** (for parsers: use same comparison function many times with different static data):



# Data Properties

Identify static data properties (with parsers in mind):

`GET\r\n` ✓ `%s` ✗  
`POST\r\n` ✓ `\v` ✗  
`RETR\r\n` ✓ `\t` ✗  
`%d` ✗

# Finding Static Data Comparisons

- 1 For each function, identify blocks that contain function calls.
- 2 Filter those blocks where the function call does not influence branching or the comparison condition is not boolean.

```
LDR R3, =request_idx
LDR R0, [R3, #(method - 0x1B030)] ; stream
BL fileno
MOV R1, R8 ; owner
MOV R2, R7 ; group
BL fchown
CMP R0, #0
BGE loc_DEB4
```



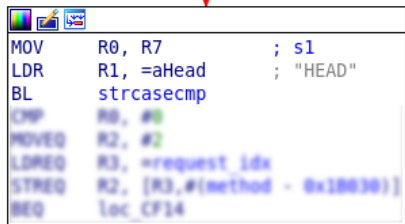
```
loc_CEA0 ; s1
MOV R0, R7
LDR R1, =aGet ; "GET"
BL strcmp
CMP R0, #0
MOVEQ R2, #1
LDREQ R3, =request_idx
STREQ R2, [R3, #(method - 0x1B030)]
BEQ loc_CF14
```

```
MOV R0, R7 ; s1
LDR R1, =aHead ; "HEAD"
BL strcmp
CMP R0, #0
MOVEQ R2, #2
LDREQ R3, =request_idx
STREQ R2, [R3, #(method - 0x1B030)]
BEQ loc_CF14
```

```
MOV R0, R7 ; s1
LDR R1, =aPost ; "POST"
BL strcmp
CMP R0, #0
MOVEQ R2, #3
LDREQ R3, =request_idx
STREQ R2, [R3, #(method - 0x1B030)]
BEQ loc_CF14
```

# Finding Static Data Comparisons (cont.)

- ③ For each argument, tag what it refers to: data section, read-only data section, other (e.g. register):

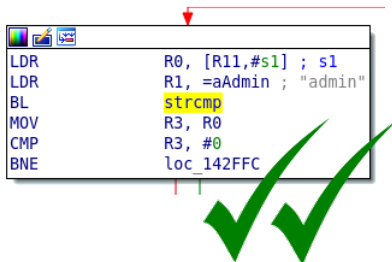
A screenshot of an assembly code window. The code is as follows:

```
MOV    R0, R7           ; s1
LDR    R1, =aHead       ; "HEAD"
BL     strcmp
CMP    R0, #0
MOVEQ  R2, #0
LDREQ  R3, =request_idx
STREQ  R2, [R3, #(method - 0x18030)]
REQ    loc_CF14
```

Annotations include a blue horizontal line above the first two lines, a red arrow pointing to the first line, and a green vertical line below the last line.

# Finding Static Data Comparisons (cont.)

- ④ Using these assignments, update likelihood of function being a comparison function:



A screenshot of a code editor window showing assembly instructions. A red arrow points to the `BL` instruction. The `strcmp` instruction is highlighted in yellow. Two large green checkmarks are overlaid on the bottom right of the code block.

```
LDR    R0, [R11,#s1] ; s1
LDR    R1, =aAdmin ; "admin"
BL     strcmp
MOV    R3, R0
CMP    R3, #0
BNE   loc_142FFC
```

# Assigning Scores to Static Data & Functions

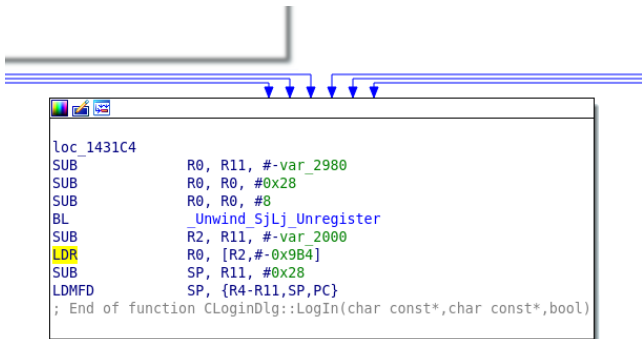


# Scoring Goals

- A means to discover those branches within each function that are dependent upon static data and assign them and the associated static data a score of relative importance in relation to other such branches within that function based upon how much unique functionality they guard.
- A function-level score that signifies which functions contain a relatively high density of decision logic that depends on comparison with static data (i.e. a large amount of their decision logic is influenced by comparison with static data).

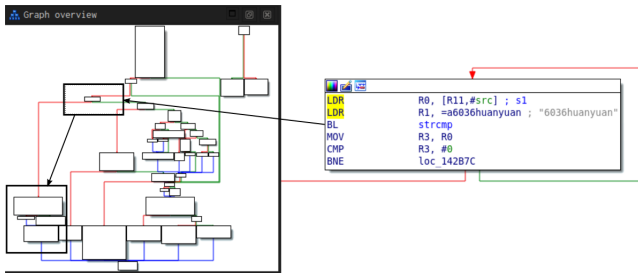
# Control Flow Properties

Minimise the score propagated from *join-points* - blocks reached by many paths:



# Control Flow Properties

Maximise score of blocks that *guard* **unique** functionality - can't be reached by any other path:



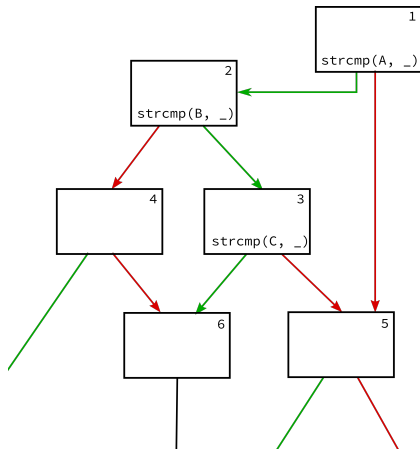
# Computation of Scores

Two stage process:

- ① Compute static data sequences: sets of sequences of static data that must be matched to reach each block.
- ② Distribute scores based upon computed static data sequences.

# Computation of Static Data Sequences

Compute sets of sequences of static data that must be matched to reach a given block:

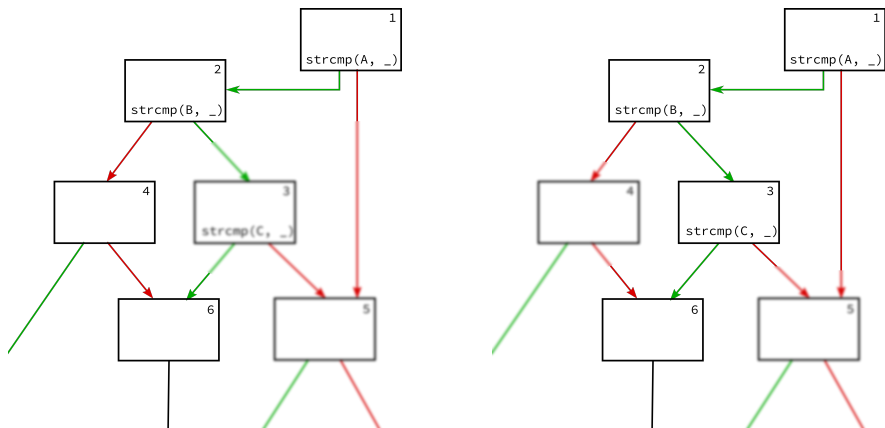


| : Comparison successful  
| : Comparison unsuccessful

- 1: {[ ]}
- 2: {[A]}
- 3: {[A, B]}
- 4: {[A]}
- 5: {[ ], [A, B]}
- 6: {[A], [A, B, C]}

# Computation of Static Data Scores

- 1 For each block's static data set of sequences, we calculate a fraction of how each element of static data impacts the reachability to that block; e.g. for block 6:



# Computation of Static Data Scores

- 1 For each block's static data set of sequences, we calculate a fraction of how each element of static data impacts the reachability to that block; e.g. for node 6:

We have:  $\{[A], [A, B, C]\}$ , so we calculate:  $A : \frac{2}{2}, B : \frac{1}{2}, C : \frac{1}{2}$ .

# Computation of Static Data Scores

- 2 We calculate two other values for the block ( $b$ ):

$$\omega(b)$$

A base score for the block

$$\frac{1}{deg_{in}(b)}$$

The penalty incurred for being reachable by multiple blocks



# Computation of Static Data Scores

- ③ ... and calculate the update to the influence of an element of static data; e.g. for  $C$ :

$$C_{score} \leftarrow C_{score} + \omega(b) \times \ln\left(1 + \frac{1}{2} \times \frac{1}{deg_{in}(b)}\right)$$

# Computation of Function Score

- The score assigned to a function is the sum of the scores assigned to the static data that influences its branching. From the previous example:

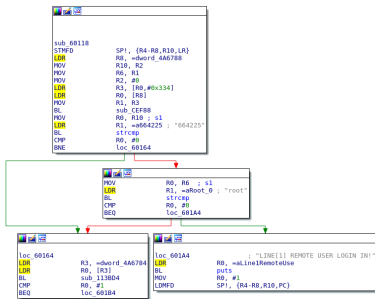
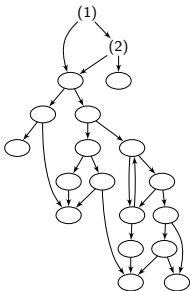
$$f_{score} = A_{score} + B_{score} + C_{score}$$

# Results & Evaluation

# Hard-coded Credentials in Ray Sharp DVR Firmware

Identification of hard-coded credential pair in Ray Sharp DVR firmware:

Comparison Function	Score
strcmp	5170.30
sub_1C7EC (strcmp wrapper)	1351.96
strncmp	1109.73
strstr	353.93
memcmp	222.00



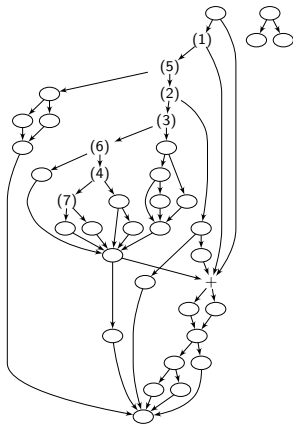
Label	Score	Static Data	Function	Depends
1	30.23	664225	strcmp	{[]}
2	2.77	root	strcmp	{{664225}}

# Hard-coded Credentials in Q-See DVR Firmware

Identification of a hard-coded credential backdoor in DVR firmware – different behaviour for each hardcoded password:

Comparison Function	Score
strcmp	1464.70
strncmp	779.33
CRYPTO_malloc (FP)	685.10
_ZNKSS7compareEPKc	376.20
strstr	306.00
strcasecmp	196.00

Label	Score	Static Data	Function	Depends
1	171.39	admin	strcmp	{{{}}}
2	58.92	ppttzz51shezhi	strcmp	{{{admin}}}
3	45.13	6036logo	strcmp	{{{admin}}}
4	42.14	6036adws	strcmp	{{{admin}}}
5	37.54	6036huanyuan	strcmp	{{{admin}}}
6	35.21	6036market	strcmp	{{{admin}}}
7	31.05	jiamijiami6036	strcmp	{{{admin}}}



# TrendNet HTTP Authentication with Hard-coded Credentials

HTTP authentication check with comparison against hard-coded credential values:

Comparison Function	Score
strcmp	1635.01
strstr	481.20
nvrnm_get (FP)	413.10
strncmp	265.45
sub_A2D0 (FP)	131.00

```

LDR R1, =aBasic ; "Basic "
MOV R2, #0 ; n
BL strcmp
CMP R0, #0
STR R0, [SP, #0x4EB0+var_4E94]
BEQ loc_C470
    
```

```

loc_C470
ADD R5, SP, #0x4EB0+s
MOV R2, #0x1F4
MOV R1, R5
ADD R0, R0, #0
BL sub_B874
LDR R2, [SP, #0x4EB0+var_4E94]
MOV R1, #0x3A ; c
STRTB R2, [R5, R0]
MOV R0, R5 ; s
BL strchr
SUBS R6, R0, #0
BEQ loc_C5AB
    
```

```

LDR R2, [SP, #0x4EB0+var_4E94]
MOV R1, R5 ; s2
LDR R0, =aEmptyuserrrrrrr ; "emptyuserrrrrrrrrrr"
STRTB R2, [R0, #1]
BL strcmp
BL R1, R0, #0 ; c
SUBS R1, R0, #0
BEQ loc_C4CB
    
```

```

MOV R0, R5 ; s
MOV R2, #0x1F4 ; n
BL memset
    
```

```

loc_C4CB
MOV R1, R0 ; s2
LDR R0, =aEmptypassword ; "emptypasswordddddd"
BL strcmp
SUBS R1, R0, #0 ; c
    
```

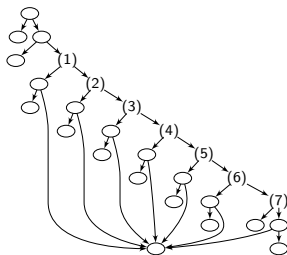
Static Data	Score	Function	Depends
emptyuserrrrrrrrrrr	132.17	strcmp	{...}
emptypasswordddddd	128.61	strcmp	{..., emptyuserrrrrrrrrrr}

# Recovery of SOAP-based Command Set

We are also able to recover the command sets of proprietary protocols, in this case a SOAP command set:

Comparison Function	Score
strcmp	380.52
safestrncmp (custom string comparison)	221.00
strstr	185.00
strcasemp	184.00

Label	Score	Static Data
1	7.64	EnableTrafficMeter
2	7.64	SetTrafficMeterOptions
3	7.64	SetGuestAccessEnabled
4	7.64	SetGuestAccessEnabled2
5	7.64	SetGuestAccessNetwork
6	7.64	SetWLANNoSecurity
7	7.64	SetWLANWPAPSKByPassphrase



- Average processing time for a binary: 1.3s.
- Some take longer - depends upon number of functions and CFG complexity:  
Q-See DVR firmware took 46.043s with 15,669 functions.



# Conclusion

- We present heuristics to automatically identify static data comparison functions effectively.
- We present complementary static data and function scoring metrics to aid in identifying hard-coded credentials and gaining insights to software functionality in a lightweight manner.
- We show our techniques are effective by discovering 3 backdoors and recovering a proprietary command set.

Questions?