Stringer: measuring the importance of static data comparisons to detect backdoors and undocumented functionality

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

DOI:
10.1007/978-3-319-66399-9_28

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

Sam L. Thomas ✉, Tom Chothia, and 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

Abstract. Finding undocumented functionality in commercial off-the-shelf (COTS) device firmware is an important and challenging task. This paper proposes a new static analysis method that measures the influence individual pieces of static data (such as strings) have upon the control flow of binaries in firmware. Our method automatically identifies static data comparison functions within binaries, then labels each function's basic blocks with the set of sequences of static data that must be matched against to reach them. Then using these sets, it assigns a score to each function, which measures the extent to which the function's branching is influenced by static data. Special keywords triggering backdoor functionality will have a large impact on the program flow. This allows us to identify three authentication backdoors – two of which previously undocumented. Moreover, we show our method is effective in aiding the recovery of both previously known and proprietary text-based protocols. We have developed a tool, `Stringer` which implements our technique; we demonstrate the effectiveness of our approach as well as its applicability to lightweight analysis by running it on a data set of 2,451,532 binaries from 30 different COTS device vendors.

1 Introduction

The current state of commercial off-the-shelf (COTS) embedded device security needs much improvement: from manufacturers deploying outdated, vulnerable software components within device firmware, to so-called *debug* interfaces being *accidentally* enabled within production versions of firmware[1]. Several backdoors,

[1] e.g., https://github.com/elvanderb/TCP-32764

undocumented commands and daemons have been reported[2,3,4,5]. The impact of these malicious or simply bad practices is exacerbated by the sheer number of devices available, with each device potentially having multiple firmware versions. Organizations handling sensitive data or critical infrastructure need a mean to determine the trustworthiness of a device before bringing it into their systems or networks. This work is currently either simply not done or is carried out manually by an expert analyst who dissembles the devices firmware with IDA Pro or similar tools. This is a very costly process that does not scale. Moreover, because the evaluation is so expensive and it needs to be done for each firmware version, it has the negative effect of motivating corporations to not update the device's firmware, leaving them exposed to known security vulnerabilities.

This work aims to reduce the effort of manual analysis by automating the identification of *interesting* code structures and functions within binaries from Linux-based embedded device firmware. This analysis is performed in a lightweight, scalable manner and is thus applicable to processing large collections of binaries from both device firmware and commodity hardware.

We say that a section of code is *interesting* when it exhibits unexpected behaviour. This behaviour is generally triggered when certain conditions are met – such as on the input of a *special* keyword. The code executed as a result of successful comparison with a *special* keyword is often not accessible by any other means, and is thus, uniquely *guarded* by that keyword.

This work automates much of the process of identifying functions that may contain functionality that is guarded by such keywords. Our method first automatically identifies the static data comparison functions within a binary. Following this, for each function we contruct the sets of sequences of static data that must be successfully compared against to reach each basic block within said function. We then use these sets to compute a score, which provides a measure of how much of a function's conditional processing is dependent on comparisons with static data.

We show that using our methods, we are able to find three backdoors, which manifest as hard-coded credential checks. In addition, we are able to demonstrate the recovery of both a known text-based protocol and a previously unknown proprietary protocol. In the case of text-based protocols, our method allows a human analyst with knowledge of known protocols to first isolate the function responsible for parsing the protocol and then identify superfluous (which are often indicative of additional, undocumented functionality) protocol messages with relative ease compared to manual analysis with tools such as IDA Pro, `strings` or `grep`.

---

[2] https://ics-cert.us-cert.gov/advisories/ICSA-13-136-01

[3] https://w00tsec.blogspot.nl/2015/11/arris-cable-modem-has-backdoor-in.html

[4] http://www.devttys0.com/2013/10/reverse-engineering-a-d-link-backdoor/

[5] https://www.sec-consult.com/fxdata/seccons/prod/temedia/advisories_txt/20160121-0_AMX_Deliberately_hidden_backdoor_account_v10.txt

1.1 Our Contribution

This paper proposes a method for lightweight large-scale static analysis of commodity embedded device firmware. We implement our techniques in a tool `Stringer`, which we use to demonstrate the effectiveness of our methods through identification of three backdoors, which we later present as case-studies in Section 5.2. Concretely, the overall contributions of this paper are:

- A set of heuristics for automatically identifying static data comparison functions.
- A metric for measuring the degree a binary's functions branching is influenced by comparisons with static data.
- The result of applying `Stringer` to a set of 7,590 firmware images, which exposes a number of backdoors. Additionally, we demonstrate how our methods can automatically identify static data processing routines. Specifically:
  - We demonstrate the recovery of the full FTP command set handled by a variant of `vsftpd` from Linksys firmware and the recovery of the SOAP-based RPC command set from a Netgear firmware's web-server.
  - We identify two previously undiscovered authentication backdoors relying on hard-coded credentials: one in a Q-See DVR and the other in a TRENDnet router.
  - We identify a third (previously reported) backdoor in firmware from Ray Sharp.

1.2 Related Work

Cojocar, et al. [6] explore the notion of a function-level metric to discover general parsing routines; they employ a fully automated approach in a similar manner to that proposed by this work; it is however, unclear of the applicability of their metric for use on a large-scale. Further, their metric relies on purely discrete counts of particular code features as opposed to more complex properties as utilised in our metric. McCabe [13] defines so-called cyclomatic complexity as a metric for computing the complexity of control flow graph (CFG); it quantifies the number of linearly independent paths through the CFG – and is hence a reasonable estimate of the branching complexity within a given CFG.

While the field of program analysis is mature, adapting traditional techniques to embedded devices is relatively new and challenging. Zaddach, et al. [19], propose a framework, Avatar, for performing semi-automated dynamic analysis upon embedded device firmware. This is done through insertion of a minimal debugger stub into the firmware itself – and hence, requires physical access to the device. A hybrid approach is taken to the dynamic analysis – relying on S^2E [5] and KLEE [3], where execution is performed both on commodity hardware through emulation and symbolic execution and in a *standard* manner upon the device itself. Similarly, FIE [11] by Davidson, et al., is a symbolic execution engine based upon KLEE, which is geared towards finding vulnerabilities in embedded microcontrollers. FIRMADYNE [4] is a framework by Chen, et al.

that like Avatar, allows for dynamic analysis via emulation of embedded device firmware; In contrast to Avatar, it does this in a completely automated manner without the need for physical access to the hardware under analysis, at the cost of only being able to analyse Linux-based firmware. Subramanyan, et al. [17] also use symbolic execution, in this case to verify the information flow properties of firmware. Firmalice [16], provides a means of identifying authentication bypass vulnerabilities, or backdoors, within device firmware, again by use of a symbolic execution engine; their techniques are however not practical for lightweight, large-scale use due to the inherent performance limitations imposed by symbolic execution (taking between 12 to 705 minutes to complete on the examples presented – which were binaries of moderate complexity). Pewny, et al. [14] propose a means of identification of bugs and vulnerabilities across different architectures and apply their technique to identify a previously known software backdoor amongst a number of firmware images from various differing vendors. Schuster et al. [15] attempt to identify potentially malicious, or anomalous codepaths in binaries on a number of architectures, including those from embedded device firmware. Their technique relies on dynamically interacting with the binary in order to explore the effect of sending specific protocol messages; thus, relies on prior knowledge of the protocol used within the binary. Thomas, et al. [18] use a hybrid approach of machine and a domain specific language to identify anomalous behaviour (including backdoors) in binaries from embedded device firmware. Both [8] and [9] provide a large-scale analysis of consumer embedded device firmware, which in the former case identified a number of known vulnerabilities within firmware; whilst the latter applies particular focus to web-frontend based vulnerabilities, the former is a more general, high-level analysis.

All of [2,10,12,7] propose methods for automatic protocol reverse-engineering. Although it is not the primary goal of our method, `Stringer` is able to extract text-based protocol messages in a lightweight, semi-automated manner.

2 Methodology

For a given binary, our method works as follows:

1. First we identify all possible static data comparison functions.
2. Then we label the basic blocks of all functions with the sets of static data sequences that must be matched against to reach them.
3. Then using the computed sets, we calculate a score for each element of static data.
4. Finally, using the scores for each item of static data we compute a score for each function.

While one approach to identifying static data comparison functions might rely on the symbol names of imported functions and look for references such as `strcmp` or `strncmp`, many binaries in firmware have their symbols stripped. Furthermore, in the case of statically linked binaries, there is no list of imported functions to extract such information. We therefore have developed a means to

automatically identify static data comparison functions which overcomes both of these problems. We do this by looking for function calls where at least one of the arguments passed is static data and the result of the function call influences control flow. Following this we rank the functions based on how they are used overall within the binary– such as the properties of the arguments they are passed and the number of arguments they are passed; Section 3 provides the complete details.

Once the static data comparison functions have been identified, we label the basic blocks of each function within the binary with a set of static data sequences. These sets dictate the sequences of static data that must be matched to reach that block. Then we calculate a score for each static data item based on how it influences the branching within the function. Finally we calculate a score for each function based on the scores assigned to the static data.

The score assigned to a function is dependent on the scores assigned to its static data. This score is used to impose an ordering of functions where those that score highly are those which contain complex decision logic that is dependent on comparisons with static data. In general, functions that implement protocol handling or contain parsing functionality are scored the highest. Further analysis of the static data and corresponding scores of those functions enables us to identify additional, undocumented functionality and (possible) backdoor functionality.

For the proof of concept tool we have developed, `Stringer`, we leverage components of `BAP` [1] and IDA Pro[6] in order to perform analysis upon concrete binaries. We rely on a number of useful components provided by `BAP`; in particular, the IL (intermediate language) it uses: BIL, its code-lifting components and the extensive set of algorithms implemented for handling graphs.

2.1 Notation

In this section we outline the notation used for the remainder of the paper. We denote an arbitrary binary as B where B is the set of its functions, denoted as $f \in B$. f_{blocks} defines the set of basic blocks for a function f. For a given block b, b_{addr} denotes the entry address of the block and b_{insns} denotes the sequence of lifted (BIL) instructions of b. $succs(b)$ and $preds(b)$ compute the set of successor and predecessor blocks of a block b.

We use the abstract notion of "sections" to denote regions of program memory that have particular properties. We assume three basic sections exist: $section_{data}$ which corresponds to the section holding data that can both be read and written and $section_{rodata}$ which corresponds to the section with constant, read-only data; we use $section_*$ to denote the union of the other two sections.

We use the notation m_k where m is a map to evaluate to the value corresponding to the key k within m. $m_k \leftarrow v$ associates the value v with the key k within m.

[6] https://www.hex-rays.com/products/ida/

3 Heuristics for Identifying Static Data Comparisons

In general, COTS Linux-based firmware images contain binaries that use a mixture of static and dynamic linking to call external library routines (such as the C standard library). Additionally, both Linux-based binaries and standalone firmware images (which in themselves can be seen as a homogeneous binary) are often devoid of symbol names. Therefore, it is both unreliable and restrictive to rely on symbol names to indicate functions used for static data comparison. To remedy this, we propose a collection of heuristics which together are able to reliably identify static data comparison functions, based on their usage within a binary. We list below the properties we expect calls to potential static data comparisons to have.

Argument references At least one function argument is either a pointer or a direct reference to either read-only program memory, or the initialised data section. From our analysis, those arguments are generally unique in functions that perform a substantial amount of static data processing.

Function arity A comparison is made between at least two items, therefore the arity, or number of arguments to a data comparison function should be at least two.

Branching properties From observation, the result of a call to a data comparison function generally influences a branching condition. Thus, one of the variables influencing the branch should be tainted by the return value of said comparison function. Further, a literal value of 0 should be compared against in the branching condition – which, in a boolean (i.e. matched/not matched) context represents *true* or *false*.

Local call frequency We observe that data processing routines such as protocol parsers generally utilise the same comparison function many times with different static data arguments as opposed to different comparison functions for each element of static data to be compared against. Therefore, we should score functions used in this way relatively highly.

Data Properties From our analysis of binaries, where comparison functions are used in protocol or message–based parsing routines we see that the static data is contained within either $section_{rodata}$ or $section_{data}$ and it is generally an ASCII-based, `NUL` terminated string. The string itself also satisfies certain properties:

- It does not contain any characters (or combination of characters) that are indicative of a format string. Concretely, we scan the string for the '%' character followed by common format directives such as 'd', 's', etc..
- It does not contain certain whitespace characters other than new line, line feed and space, such as: tab ('\t'), vertical tab ('\v'), etc. or those characters that are used as control characters.

3.1 An Algorithm for Finding Static Data Comparisons

We now outline our algorithm for identifying static data comparison functions. For each function in the binary, we identify all blocks that contain function

calls. Of those function calls, we filter out those that don't influence branching conditions where that condition is a comparison against 0. Of the remaining function calls, we analyse the arguments passed. For those arguments we define two cases: the ideal case and a "catch-all" case.

The ideal case occurs when the function invocation involves at least two arguments where one of those is a reference to static data that conforms to the properties outlined in Section 3. In addition to those constraints upon the data references, at least two of the arguments should not be register-based constants such as integers or floating point numbers (that are not also address references to $section_*$). We impose this restriction as we expect a comparison to make two references to data – one static, the other dynamic. The general "catch-all" case occurs when at least two of the arguments identified do not reference constant data.

The result of applying our algorithm is a set of comparison functions along with a score representing the likelihood that that function is a comparison function.

```
1: function computeHeuristicScores(ς, δ, μ+, α-, α*, B)
2:     ν ← {}
3:     for each f ∈ B do
4:         ν' ← {}
5:         for each b ∈ {b | b ∈ f_blocks ∧ branchesOnCall(b) ∧ branchesOnZCmp(b)} do
6:             args_data ← ∅, args_rodata ← ∅, args_other ← ∅
7:             for each arg ∈ dependentArgs(b, 3) do
8:                 if arg ∈ section_rodata then
9:                     arg_rodata ← arg_rodata ∪ {arg}
10:                else if arg ∈ section_data then
11:                    arg_data ← arg_data ∪ {arg}
12:                else if arg ∉ section_* then
13:                    arg_other ← arg_other ∪ {arg}
14:                end if
15:            end for
16:            addr ← f_addr
17:            if |arg_rodata| + |arg_data| + |arg_other| ≥ 2 ∧ containsIdealSD(args) then
18:                ν'_addr ← ν'_addr + ς
19:            else if |arg_data| + |arg_*| ≥ 2 then
20:                ν'_addr ← ν'_addr + δ · ς
21:            end if
22:        end for
23:        ν ← mergeScores(ν, applyRewards(applyPenalties(ν', α-, α*), μ+))
24:    end for
25:    return ν
26: end function
```

Fig. 1. Algorithm to compute heuristic scores

We use the notation ν_{addr} to represent the heuristic score for the function with entry point at address $addr$. ς represents the value to increase ν_{addr} by when a block satisfying the ideal case is encountered. When the ideal case is not encountered, we use a multiplier δ to scale ς such that $0 < \delta \leq 1$ prior to incrementing.

After processing the function f, local scores for each possible data comparison function are merged into a global map of scores. Prior to this merge, we apply two modifiers as *rewards* and *penalties*: we scale up the score of the suspected

static data comparison function with the highest number of call-site occurrences within f by a constant μ_+, where $\mu_+ \geq 1$ (local call frequency), we scale down the score of every function h that references the same static data multiple times by α_-, where $0 < \alpha_- \leq 1$ (argument references). We apply further scaling of α_- by α_* which is raised to the number of non-unique data references n, used as arguments to h. That is, if h has the address h_{addr}, $h_{addr} \leftarrow h_{addr} \cdot \alpha_- \cdot \alpha_*^n$.

The algorithm in Fig. 1 outlines the computation performed to calculate the heuristic scores for all possible comparison functions within B. For brevity, the algorithm makes reference to a number of functions: *containsIdealSD* which evaluates to *true* if at least one of the expressions in the set passed as an argument satisfy the aforementioned data constraints; *branchesOnCall* evaluates to *true* if any variable in the conditional expression of the block is tainted by the last function invocation within the block; *branchesOnZCmp* evaluates to *true* if the conditional expression of the block depends on a comparison with 0 (or a semantically equivalent boolean comparison); deg_{in}/deg_{out} evaluate to the number of incoming/outgoing edges from the block; *dependentCall* evaluates to the function that would cause *branchesOnCall* to evaluate to *true*; *dependentArgs* evaluates to a map of at most n expressions that correspond the the arguments passed to the function call that *dependentCall* evaluates to. *applyPenalties* and *applyRewards* perform the previously outlined score modifications. While *mergeScores* merges the locally computed scores (on a function-level basis) into the global map of scores. For each function, we use a *local* map, ν' to store the computed values for that function prior to merging into μ. In our implementation, the variables are assigned values based on small-scale experiments; in all cases (50 binaries) the C standard string comparison functions are identified.

4 A Metric For Scoring the Importance of Code

This section defines our metric that is used to determine the degree to which a given piece of static data influences the execution of a function. Our metric provides:

- 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).

4.1 Requirements of the Metric

Our metric's goal is to score functions that contain decision logic that depends upon static data comparisons where that static data tends to uniquely isolate functionality within a function highly.

We assign a score to a given element of static data depending on how it isolates functionality within a function's CFG – if the only way to reach part of the CFG is via successful comparison with an item of static data, that static data shall score highly. The scores for each piece of static data are computed based on the successor blocks following the use of that static data as an argument to a comparison function.

Within CFGs there are a number of possibilities how to propagate values; we base those on observations of basic block properties:

Number of incident blocks A block that has many incident edges can be considered a *join-point*, that is its functionality is of less importance to the functionality of a single isolated code path as it is reachable by many paths throughout the function. Thus, the influence that such a block should have upon its predecessor blocks should be distributed relative to the number of incident edges (i.e. $deg_{in}(b)$).

Branches as "guards" of functionality We associate the code of successor blocks with the branch that is *guarded* by the branch condition evaluating to *true*. For any given static data comparison, the degree it divides the overall CFG along the branch which is followed when the comparison is *true* is a general indicator of the importance of that string comparison. The static data which guards large amounts of functionality should have a higher score than that that does not. Applying this directly to a scoring metric however would cause those comparisons that happen first within the CFG to be assigned significantly higher scores than those that happen later. A notion of dependence and a means to be able to diminish the value of that dependence in a manner proportional to the distance along the path of static data comparisons required to reach a given point within the CFG should overcome this.

4.2 Definition of the Metric

The calculation of our metric is performed as a two stage process: we first construct sets of static data sequences at each block within the CFG; then using these computed sets we perform a single pass over all blocks within the CFG and assign a score to each branching block that contains a comparison with static data.

The computed static data sequence set for a block represents all possible positive static data comparisons taken to reach that block. For instance in Fig. 2, if we consider both nodes 1 and 2 static data comparisons where the branches from $1 \rightarrow 2$ and $2 \rightarrow 3$ are taken if the comparisons at 1 and 2 evaluate to *true*. Then the sets we compute are as in Fig. 3. We use the notation s_i to represent the static data compared against at node i.

We compute the sets by using the algorithm in Fig. 4; it is applied to each block until the computed static data sequences reach a fixpoint. The notation $+\!\!+$ is used to denote the concatenation operator on sequences, $branchData(b)$ computes the static data compared to at block b, $branchesOnStaticData(b)$ evaluates to *true* if the block b's branching is dependent upon a comparison with static data . Loops are ignored when determining if a fixpoint is reached;

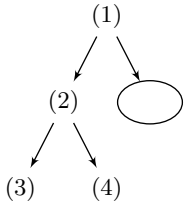

Fig. 2. Example CFG with static data comparisons

Label	Computed string sequence set
1	$\{[]\}$
2	$\{[s_1]\}$
3	$\{[s_2, s_1]\}$
4	$\{[s_1]\}$

Fig. 3. Computed string sequence sets for Fig. 2

this ensures termination and avoids the construction of sequences with repeated sub-sequences.

In COMPUTESDS(B), if after iterating over all of $preds(b)$, b_s is equivalent to $\emptyset$, then we set b_s to $\{[]\}$. This represents there is no known path to reach b that is dependent upon successful comparison with static data.

We compute the scores for each block using the algorithm in Fig. 5. In our implementation, the function to compute basic block complexity ($\omega(b)$) evaluates to $|b_{insns}|$ – the number of lifted (BIL) instructions within the block. The algorithm takes each previously computed static data sequence and computes a score for each block that is associated with a static data comparison. For each block, we take the set of static data sequences S and update the score of each element of static data found within those sequences. The final result is a map of static data and corresponding scores.

```
1: function computeSDS(b)
2:     for each p ∈ preds(b) do
3:         if branchesOnStaticData(p) then
4:             s_p ← branchData(p)
5:             b_s ← b_s ∪ {S_i ++ s_p | S_i ∈ p_s}
6:         else
7:             b_s ← b_s ∪ p_s
8:         end if
9:     end for
10:    if b_s = ∅ then
11:        b_s ← {[]}
12:    end if
13: end function
```

Fig. 4. Algorithm to compute static data sequences

```
1: function computeScores(ω, f)
2:     M ← {}
3:     for each b ∈ f_blocks do
4:         S ← b_S, baseScore ← ω(b), numChains ← |S|, countMap ← {}
5:         for each S_i ∈ S do
6:             for each s ∈ S_i do
7:                 countMap_s ← countMap_s + 1
8:             end for
9:         end for
10:        for each s ∈ countMap do
11:            occScale ← countMap_s / numChains
12:            M_s ← M_s + baseScore × ln(1 + occScale × 1 / deg_in(b))
13:        end for
14:    end for
15:    return M
16: end function
```

Fig. 5. Algorithm to compute scores

For each block, the computation is approached in two sub-phases: the first constructs a mapping of static data to the number of times said element of static data s occurs within the sequences within the set of static data sequences. This count is used to determine a scaling factor which is a fraction of the total number of sequences and the count of those that s is present in. This value is representative of how much the reachability of a given block depends upon successful comparison with a given element of static data: if the static data *has* to be matched to reach the block then the fraction shall be equivalent to 1. Following this, for each element of static data within the previously discussed map, we compute the sum of the base score assigned to the block (computed by $\omega(b)$) scaled by the mentioned scaling factors and the current score assigned.

An additional scaling factor is also computed which is equivalent to the inverse of the number of incident edges to the block: i.e. $\frac{1}{deg_{in}(b)}$.

The previous two phases compute a block-level score. We define the *importance* of a function as the sum of scores assigned to each element of static data. This allows us to identify functions where decision logic is largely influenced by comparisons with static data.

5 Results

We have implemented the aforementioned heuristics and metric within our tool, `Stringer` which automates the entire analysis process: firmware acquisition, unpacking and report generation. In this section we discuss the outcomes of running our tool upon a firmware collection totalling 7,590 successfully unpacked firmware images, equating to 2,451,532 individual binaries.

5.1 Experiment Methodology

While `Stringer` automates the majority of the analysis process, a degree of manual intervention is required to discern the most *interesting* binaries from the processed data-set.

First we use a web and FTP crawler to download 15,438 firmware images from 30 different vendors. For each firmware image downloaded, we attempt to extract its filesystem using existing tools: `binwalk`, `sasquash` and `jefferson`. Our resulting data-set consists of 7,590 successfully unpacked firmware images. With the resulting filesystems, we search for binaries and each of the 2,451,532 binaries are passed to `Stringer`, which generates a report for each.

Then we perform semi-automated analysis of the generated reports. We attempt to discover routines handling common protocols; for this we devise some simple models of what static data we expect to be grouped together within a single function. For instance, for a web-server we expect the terms `GET` and `POST` and possibly, `PUT`, `HEAD` and `DELETE`. Additional static data found within these routines is further analysed manually. In addition to this, we use `grep` to search the reports for *interesting*, "low-hanging fruit" by searching for terms such as `admin`, `Administrator` and `root`. Our report format details the highest scoring

functions along with the associated static data, which is tagged with the score it contributes to the overall score for the function.

Once an interesting binary is identified, we perform manual analysis using IDA Pro. The standard manual analysis process is aided by the fact the functions and strings of interest are available from the generated reports and so anomalous functionality such as backdoors or undocumented commands can quickly be checked for, and confirmed.

Due to the modest amount of backdoors publically available that are both present in embedded device firmware and also backdoors that are of the class that can be detected by `Stringer`, calculating the FP rate of our technique is infeasible.

5.2 Case-studies

Due to the large amount of binaries processed as part of the analysis, we present a selection of case-studies. Each case-study follows a similar form: we first present the scores and ranking of possible comparison functions as computed by application of our heuristics. Then we present *interesting* functions identified by application of our metric.

5.2.1 Identification of the FTP command set The `vsftpd` FTP server, shared amongst numerous Linksys device firmware images provides a clear example of the effectiveness of our approach. The binary analysed, contains a total of 600 functions, uses static linking and is stripped of symbol information. Our heuristic identifies 44 potential comparison functions; those ranked highest are: `sub_10814` (394.84), `sub_1622C` (35.00), `sub_10754` (27.20), and `sub_139FC` (12.20). As `vsftpd` is open-source software, we are able to discover that `sub_10814` corresponds to the function `str_equal_text` – a string equality check for the `vsftpd`'s custom string implementation.

The metric finds the highest ranking function to be the main protocol parsing routine: `sub_C4F0` which is assigned a score of 942.08 (and corresponds to `process_post_login`). The FTP command set handled by `vsftpd` is extensive; we therefore omit the specific output of the tool and associated CFG due to its size.

The metric scores for `sub_C4F0` group the protocol messages; the uniformity that is apparent reflects the implementation of the state-machine used to handle connections. That is, following matching the input with a protocol message; a secondary function is called which handles further processing of the input or the functionality of a specific command. The group of highest scoring protocol messages (`HELP`, ...) have scores of 16.00, while the lowest (such as `PROT`) score 8.03. The largest group of commands with uniform scores of 10.00 contains the core command set (`STOR`, `RETR`, `PASV`, `PORT`, `LIST`, `QUIT`, ...).

5.2.2 Hard-coded credential backdoor #1 In a number of firmware images for QSee DVR products, `Stringer` identifies numerous hard-coded credentials – to the best of our knowledge, this backdoor is previously undiscovered – which provide differing levels of access to the device. The binary used for this case study, `td3520` contains a total of 15,669 functions and is statically linked. The heuristics identify a possible 911 comparison functions, those that are ranked highest are: `strcmp` (1464.70), `strncmp` (779.33), `CRYPTO_malloc` (685.10) (from the statically linked OpenSSL library), `_ZNKSs7compareEPKc` (C++'s string equality operator) (376.20), `strstr` (306.00) and `strcasecmp` (196.00). All but one of those functions is a static data comparison function; a single false positive, `CRYPTO_malloc` is identified due to its usage patterns being almost identical to that of an expected comparison function.

We identify the third highest ranked function by our metric as `_ZN9CLoginDlg5LogInEPKcS1_b` (scoring 421.38) which contains a hard-coded credential checking routine. Fig. 6 shows the scores and sets of static data sequences of the static data extracted from that function, while Fig. 7 shows the simplified CFG with static data labelled using those in Fig. 6.

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`]}

Fig. 6. Scores for `_ZN9CLoginDlg5LogInEPKcS1_b`

We observe that successor nodes that are dependent on the highest ranked static data (`admin`) follow from the left branch of the comparison node. All other static data comparisons are dependent upon a successful comparison with `admin`. The static data ranked as second most important isolates most unique functionality relative to the other identified static data (in this case that past the node labelled with a +: this is the functionality associated with a successful login with administrative credentials.

This binary was first located by searching through the reports generated by `Stringer` for common privileged usernames: namely, `admin`; verification of the backdoor was performed manually using IDA Pro.

5.2.3 Hard-coded credential backdoor #2 `Stringer` also finds a hard-coded credential check within firmware from Ray Sharp – a popular CCTV DVR vendor. The binary containing the backdoor, `raysharp_dvr` contains a total of 7,605 functions, is dynamically linked and stripped of local symbol names. The heuristics reveal the highest ranked comparison functions to be those from the C standard library: `strcmp` (ranked highest) (5170.30), `strncmp` (1109.73), `strstr`

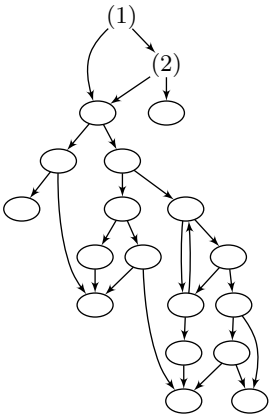

Fig. 9. Ray Sharp hard-coded credential check

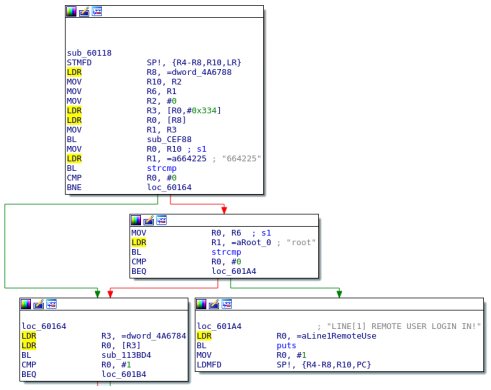

Fig. 10. IDA Pro CFG snippet of the Ray Sharp backdoor

(353.93) and `memcmp` (222.00). Additionally, `sub_1C7EC` (1351.96) is ranked second – which from manual analysis with IDA Pro is identified as a wrapper around `strcmp`.

The functions our metric scores highest consist of complex parsing routines – indicated by their relatively high scores compared to other *interesting* functions identified. `sub_60118` contains the functionality responsible for the backdoor. Fig. 9 details the CFG of the function along with the scores assigned to the username and password combination and Fig. 8 shows the scores as assigned by the metric as well as computed static data sequence sets. Fig. 10 shows an IDA Pro CFG snippet of the backdoor.

This binary was identified by searching the logs for common usernames that are associated with privileged user accounts: in this case, `root`.

A posteriori research online shows that (in contrast to the other case studies) this discovery was not original. The backdoor has been previously documented[7] and is present in a multitude of devices from

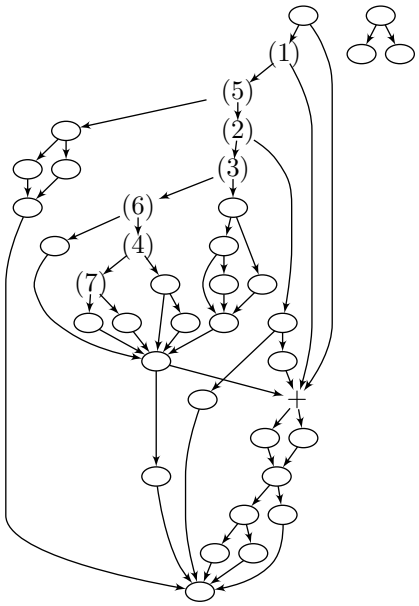

Fig. 7. CFG for `_ZN9CLoginDlg5LogInEPKcS1_b`

[7] https://community.rapid7.com/community/metasploit/blog/2013/01/28/ray-sharp-cctv-dvr-password-retrieval-remote-root

many vendors: Swann, Lorex, URMET, KGuard, Defender, DEAPA/DSP Cop, SVAT, Zmodo, BCS, Bolide, EyeForce, Atlantis, Protectron, Greatek, Soyo, Hi-View, Cosmos, and J2000.

5.2.4 Additional functionality within standard protocols In the bundled web-server found within the firmware of a number of TRENDnet devices, `Stringer` identifies a hard-coded credential pair within the routine handling basic HTTP authentication. The comparisons are performed via standard string comparison (`strcmp`) – which is ranked by the heuristic as the most likely static data comparison function. It identifies 40 such functions out of a total of 391 functions within the entire binary. `strcmp` is ranked highest by a large margin with a score of 1635.01, followed by `strstr` (481.20), `nvram_get` (413.10), `strncmp` (265.45) and `sub_A2D0` (131.00). `sub_A2D0` provides a wrapper around `hsearch_r` – a lookup function for hash tables, evaluating to 0 on failure. Both `nvram_get` and `sub_A2D0` may be regarded as false-positives: the former provides a lookup of the embedded devices NVRAM (Non-Volatile RAM – an area on the device usually used for storing configuration that can persist across device restarts).

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

Fig. 8. Scores for `sub_60118`

The additional functionality is embedded within the eighth highest scoring function – `sub_B958`, with a score of 827.99. Whilst validating the credentials for HTTP basic authentication, an additional code path checks for the hard-coded username/password pair: `emptyuserrrrrrrrrrrr` and `emptypassworddddddddd` both via `strcmp`, which score 106.00 and 103.47, respectively and rank as the second and fourth most important strings. The most important string is the string comparison to detect if basic authentication is being used and scores 151.84. We omit a diagrammatic representation of the CFG due to space considerations.

This binary was located by matching the logs against the common authentication header strings used in HTTP authentication. To the best of our knowledge, the hard-coded credentials have not been previously documented. Again, manual analysis was performed using IDA Pro.

5.2.5 Recovery of SOAP-based protocol command set The firmware from a number of Netgear devices contains a web-server, `mini_httpd` that uses SOAP[8] for RPC. The binary contains 331 functions in total, 60 of which are identified as possible static data comparison functions by the heuristic. Those that are ranked highest are a combination of standard functions from the C standard library: `strcmp`, `strstr` and `strcasecmp` scoring 380.52, 185.00 and 184.00, respectively as well as a custom comparison function (ranked second): `safestrcmp` scoring 221.00.

[8] https://www.w3.org/TR/soap12/

The function ranked highest by the metric is `handle_request` (scoring 952.91) – which processes the HTTP protocol. Ranked second is `do_file` scoring 486.47, while the *main* (scoring 449.55) function is ranked third – which provides argument parsing for the binary. `soap_parent_ctrl_handle` is assigned a score of 328.75 and is ranked fourth; it handles the processing of the SOAP command set. This function exemplifies the effectiveness of `Stringer` in extracting protocol command sets that are previously unknown to the analyst. The scores assigned to individual command strings within the function are uniform. Fig. 12 is a fragment of the CFG for `soap_parent_ctrl_handle` and Fig. 11 contains the scores of the static data present in that fragment.

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`

Fig. 11. Selection of static data from `soap_parent_ctrl_handle`

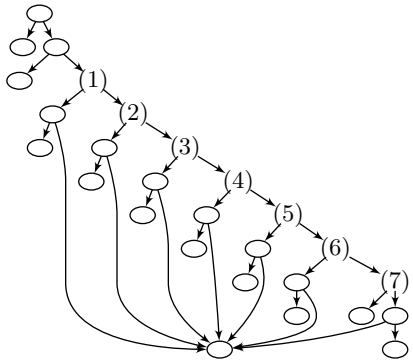

Fig. 12. CFG fragment for `soap_parent_ctrl_handle`

The command set was discovered by searching logs for web-server related protocol strings, in this case: `GET`. This string existed (amongst other HTTP commands) in the higher scoring function `handle_request`; `soap_parent_ctrl_handle` was located by looking at other high ranking functions within the binary.

5.3 Performance

On average a firmware image contains a total of 379 binaries; with each binary taking 1.31s to process. Larger binaries, with a greater number of functions or larger CFGs take considerably longer; though, the performance is still acceptable for large scale analysis. As a concrete example, the binary `td3520` (Section 5.2.2) which contains 15,669 functions, took 46.043 seconds. A significant portion of the total runtime for `Stringer` is due to the invocation of IDA Pro to export data required for CFG recovery. The total time taken to invoke IDA takes on average 11.26% of the total execution time. Processing this data takes on average 0.63% of the total time. The remainder of the time is due to computation of the heuristic and metric scores.

5.4 Comparison with Naïve Techniques

The techniques we have shown improve upon existing techniques for identification of interesting static data. Past work has used a combination of linux functions `strings`, to extract stings from binaries, and `grep`, to find interesting terms, or more advanced processing methods such as using IDA Pro with `IDAPython` to export static data coupled with function names with further processing performed using `grep`. Neither of these existing tool combinations provide any indication of the importance of a given piece of static data in relation to any other. Furthermore, neither provide a means of ranking the importance of functions in relation to how much of their conditional processing is influenced by static data. The lack of both of these properties limit the effectiveness of these methods, meaning that a large amount of manual analysis, and some luck are required when analysing large pieces of firmware. Moreover, these techniques only scale to locate functionality based upon *known* protocols or easily recognisable strings.

6 Conclusion

We have presented a novel approach to identify static data comparison functions within binaries, which when combined with our function-level scoring metric, as demonstrated, is effective in discovering undocumented functionality and recovery of text-based protocol messages and commands. In the case of the former, we have identified a three instances of authentication backdoors in commodity firmware images from a number of vendors. Our approach is shown to be suitable for large-scale analysis – with methods presented for reducing the effort required by a human analyst processing the resulting data. With our technique we are able to isolate functions of interest ranking them within the first tens of functions as opposed to an analyst having to trawl through potentially thousands of functions. A concrete example of this is from our case-study in Section 5.2.2, whereby the most interesting function for an analyst is ranked as third most important out of $15,669$ functions.

Our approach improves on existing large-scale analysis methods upon embedded device firmware by performing more complex static analysis – that considers the control-flow properties of code – as opposed to propagating known bitstring patterns over the data-set. Moreover, we introduce a new means of identifying potential functionality for binary analysis – which is applicable beyond binaries within embedded device firmware.

References

1. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Computer Aided Verification*. Springer, 2011.
2. J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, CCS 07. ACM, 2007.

3. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI 08. USENIX Association, 2008.
4. D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 16, 2016.
5. V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI. ACM, 2011.
6. L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti. PIE: Parser Identification in Embedded Systems. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015.
7. P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *2009 IEEE Symposium on Security and Privacy*.
8. A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.
9. A. Costin, A. Zarras, and A. Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *11th ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, ASIACCS 16, 2016.
10. W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, CCS 08. ACM, 2008.
11. D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.
12. Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution. In *NDSS 08*, 2008.
13. T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 1976.
14. J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy*, 2015.
15. F. Schuster and T. Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security*, CCS 13. ACM, 2013.
16. Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 15, 2015.
17. P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung. Verifying information flow properties of firmware using symbolic execution. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016.
18. S. L. Thomas, F. Garcia, and T. Chothia. HumIDIFy: A Tool for Hidden Functionality Detection in Firmware. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, DIMVA 17. Springer, July 2017.
19. J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 14, 2014.