Implementing Tools for Visual Reverse Engineering

and Breaking Flash Based CAPTCHAs

Wannes Rombouts-Fockaert
Student ID: 13909917
wr43@kent.ac.uk

School of Computing, University of Kent

September 13, 2014

Abstract

Understanding the layout of unknown files can be a difficult task, especially with traditional tools such as hex editors. Visual Reverse Engineering has emerged as an alternative way to approach binary data through different visualization and analyses. In this paper we present the results of the implementation of several state of the art algorithms in an open source tool. We also show our work can be used to recognize a number of typical data types. We where able to reproduce some results claimed by the authors of Cantor Dust which has never been released to the public. The second part of this paper shows general weaknesses in the implementation of flash based CAPTCHAs and presents three attacks that where successfully used to break the Dracon CAPTCHA. We focus on passive OCR based attacks exploiting weaker frames in an animated CAPTCHA, active attacks tampering with the compiled flash file and attacking the design flaws in the integration of the flash object in a web page. Those attacks, while applied to an example in this paper aren't specific to it and should be taken into account when designing interactive CAPTCHAs.

1 Visual Reverse Engineering

1.1 Introduction

The original subject of this research was supposed to be looking into the many vulnerabilities that have been found in firmwares of consumer electronics in recent years. When analyzing unknown firmwares there was a lack of tools that could help with finding structure in the opaque binary file formats. Cantor Dust [1] seemed like a promising work but isn't available to the general public at this time. As part of my M.Sc project I wrote binglide which is a tool using some of the newer methods of binary data visualization. The remainder of this introduction will present some of the existing tools and techniques. The algorithms used by binglide and there implementations are described in sections 1.3 and 1.4. Our results and a description of how binglide can help recognizing typical data types and analyze unknown file formats can be found in Section 1.5.

A note on Figures

Most of the images in this paper have been inverted. This has been done because the originals where low contrast on black backgrounds and didn't render well in print. Original versions can be found on binglide's github [2].

1.1.1 binwalk

Binwalk [3] is now one of the de facto tools for analyzing unknown binary files. It is an open source tool for windows. It scans a file for known headers and magic numbers and provides a listing of all sections of the file matching a recognized file format. This is very use full when dealing with well behaved files but breaks when the file has been intentionally obfuscated or even accidentally corrupted. Other features include an entropy graph showing regions of the file with low or high entropy which can be used to identify encrypted or compressed regions padding, or objects such as encryption keys.

1.1.2 binvis

Binvis [4] is one of the first visual reverse engineering tools. It uses visualizations such as dotplots [5, 6] and introduces bigram histograms as a tool for analyzing unknown binary data.

During their talk at BlackHat [7] the authors of this tool presented an interesting table categorizing various reverse engineering tools according to if they where general purpose or meant for a precise application and the insight they provided. They concluded that all general purpose tools such as strings, grep and hexeditors provided lower insight into the data. Tools providing more insight, for example Ida Pro or BinDiff where tended to be very application specific. The authors claimed visual reverse engineering tools such as binvis might lead to general purpose tools that offer higher insight.

1.1.3 Cantor Dust

Cantor Dust [1] has been presented at some conferences [8] but isn't publicly available as of the time of writing. It is also a windows tool but offers a much more interactive user experience than binvis. The user can dynamically select parts of the file to analyze and switch between different views. Cantor Dust extended the idea of bigrams to a three dimensional representation of trigrams that can be projected in various shapes.

It offers many other views such as entropy, interpretations of the raw data in different encoding with the ability to specify the number of bytes per pixel, and generic callgraph generation for binary files.

This tool offers a very user friendly interface which binvis lacks but is important for visual

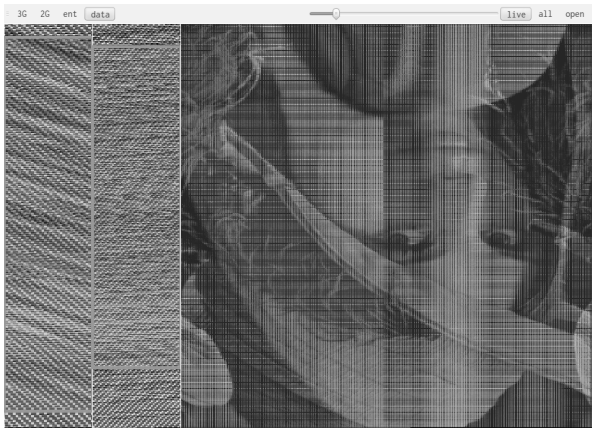

Figure 1: Binglide user interface, showing the data view when looking at a bmp file.

reverse engineering to be truly effective.

1.2 binglide

binglide, the opensource tool developed as part of this research implements four different views. A plain data view similar to what one would get if the data was considered as an image using one byte per pixel, an entropy visualization and a two-dimensional and three-dimensional representation of bigrams and trigrams respectively.

The user interface can be seen in Figure 1, the user can use the two panes on the left to zoom-in onto a specific region of the file that they want to analyze. The file being looked at in this example is a bmp file, by playing with the region selection we can get a glimpse of the image in binglide, this without to tool knowing what a bmp file is and only displaying the raw data it contains.

Binglide was developed using python because it is a very expressive language that allows for fast prototyping. The user interface uses QT through the pyqt module. Visuals are generated using the pyqtgraph modules which adds support for OpenGL to pyqt and provides a large panel of ready to use widgets for displaying plots and images.

Another advantage of python is its portability, binglide has been successfully tested on Windows, Mac and Linux platforms.

Simply relying on the raw speed of other languages wouldn't be a solution that scales well, using python with some optimizations we managed to have very reasonable speeds on files under 100Mb which was deemed sufficient for this first version intended as a proof of concept.

1.3 Entropy

The entropy of a file often is an indication of what type of data it contains, this is also true and more useful for parts of the file. For example a high entropy region in a malware is likely to contain encryption keys or encrypted code while lower entropy might reveal the location of the decryptor.

1.3.1 Implementation

In order to have a visual representation of the local entropy in a file we used Shannon entropy over a sliding window as suggested by Cortesi [9].

Shannon entropy is the information entropy of a dataset. This is expressed in bits per symbol and can be calculated as follow: $H(S) = -\sum_{i=0}^{n} P(s_i) \log_2(P(s_i))$ where P is the relative frequency of each character which can be obtained using a histogram.

For the implementation to be effective we added some optimizations to the naive algorithm which would compute the window's entropy for each byte. Instead we maintain a queue of the values that fall in the window. This allows us to subtract what was contributed by the byte that exits the window and add the value for the byte that is entering it, without the need to recompute the whole Shannon entropy. In addition to this, since we only have 256 different symbols we precomputed all

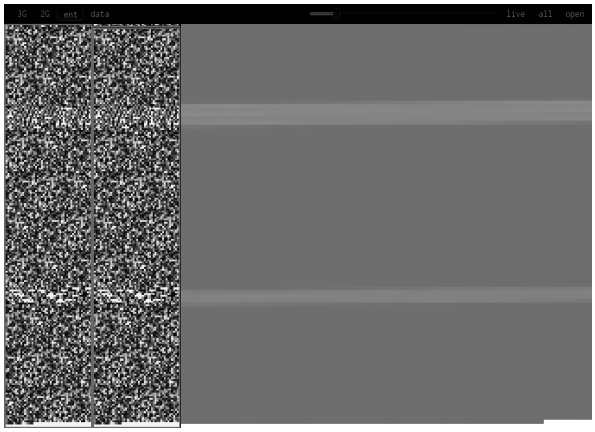

Figure 2: Entropy visualization of shellcode hidden in random noise.

the possible logarithms which significantly increases the speed at which python can perform the calculations. Still this isn't fast enough on larger files for real time use so we used the just in time compiler from the numba framework [10] with great success.

1.3.2 Visualization & Results

The visualization for the entropy is simply a mapping of the local entropy value to a color. It is very simple and effective in helping the user identify higher entropy regions.

Figure 2 shows the entropy view for two small shellcodes that are hidden in a lot of random noise. The first one might be noticeable by looking at the plain data visualization on the left but the second one is much harder to spot without the entropy visualization.

1.4 N-Grams

Using the frequency of bigrams for analyzing data is not a new idea, It has been used in cryptography for a long time. Its application to reverse engineering however is more recent and was popularized by binvis. Cantor Dust extended the idea to representing trigrams in three dimensions and we implemented the same algorithms.

1.4.1 Computation

In order to implement this an algorithm has to be used that can efficiently compute a histogram of an arbitrary section of the file. Doing the whole computation when needed is possible for small sections but is very slow when that section is large.

An ideal solution for speeding up this computation would be the use of a binary tree of histograms covering the whole file. With this data structure computing the histogram of an arbitrary section of the file is a simple matter of adding up the histograms that compose it which is fast. The problem with this data structure is that it tend to be very large and RAM quickly becomes a problem on the large files where such a speedup would be needed.

As an alternative we split the files in chunks and pre-compute a histogram for each one. This is similar to the tree but consumes less RAM at the cost of being less efficient.

1.4.2 Visualization

For the visualization itself simply plotting the raw values for each ngram in a n-dimensional space gives results similar to binvis but isn't as beautiful and readable as what we can see on Cantor Dust in the demonstrations. Cantor Dust not being released and no details being available about the way it renders its visualizations we had to improvise.

The algorithm we use sorts all the ngrams according to their value which is how many time they occur. For each ngram the value that is being rendered is the sum of all lower values relative to the total sum of values. $P(G_i) = \sum_{j=0}^{i} V_j / \sum_{k=0}^{256^n} V_k$ where G_i is the i^th ngram in the sorted list of all ngrams. The total sum of values is actually $N-n+1$ where N is the total

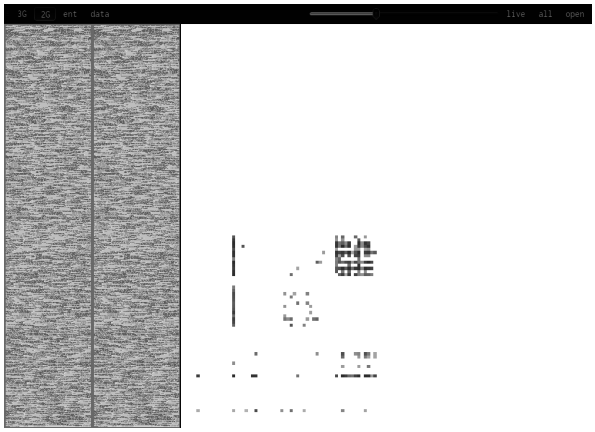

Figure 3: english text bigrams

Figure 4: 8bit human speech bigrams

file size, therefor the value being rendered can be expressed as $P(G_i) = \sum_{j=0}^{i} V_j/(N-n+1)$

This visualization seems to come very close to what Cantor Dust uses and can be seen in Figures 3 to 14.

1.4.3 Results

When observing different file types we notice that similar data types tend to produce very similar patterns. We where also able to verify the claim made by the authors of Cantor Dust that different data types with a similar purpose often also had the same signature. For example we can confirm that byte code for different processor architectures has the same visual patterns. More on this in section 1.5 where we provide more details about different data types and screen-shots of the visualizations.

1.5 Analysis

In this section we present some of the most common datatypes and what they look like in binglide. Then we show a practical example of how the tool can be used to get a very high level overview of a binary dump of an unknown firmware.

1.5.1 Text

Figure 3 is a 2D representation of the histogram of bigrams in english text. We can see a square representing the lowercase ASCII characters that follow each other. Slightly to the left we see some traces of uppercase followed by lowercase and beneath that another, more subtle, square representing upper case words. The rest is mainly characters followed by spaces or other punctuation. It is interesting that we can see that characters such as commas and periods always follow lowercase characters and are almost always followed by spaces.

1.5.2 Sound

Figures 4 and 5 show the bigrams for human speech and music respectively. In both cases we can see a very uniform distribution around the diagonal indicating very low variations from one byte to the next. This is consistent with "natural" data such as sound or raw images. The very low range used by human speech is reflected in the representation and gives a very good idea of why compression algorithms are so successful on it.

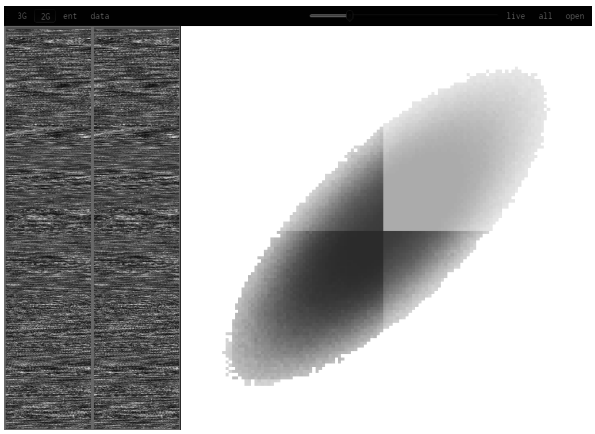

Figure 5: 8bit music bigrams

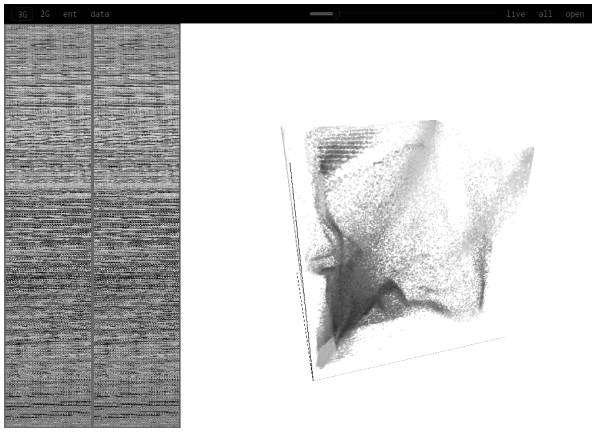

Figure 6: RGB image trigrams

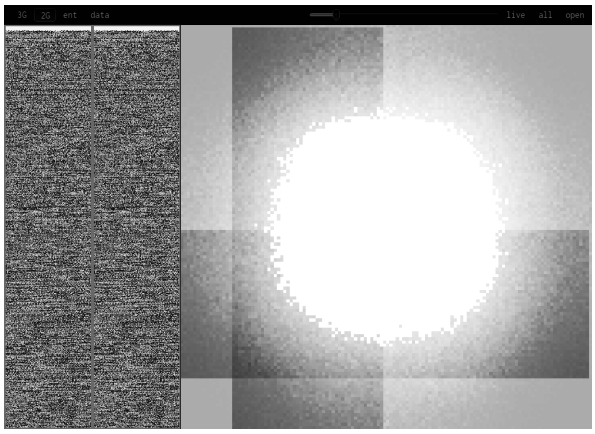

Figure 7: 16bit encoded music bigrams

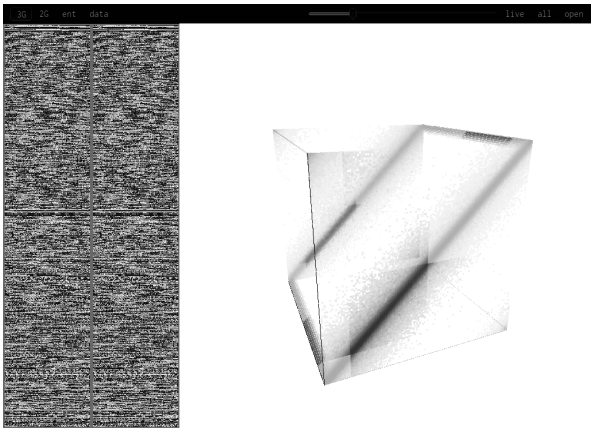

Figure 8: 16bit encoded speech trigrams

1.5.3 Images

Images have the same diagonal shape as sound but are much more organic as can be seen in Figure 6). This is because most images are not composed of only one element but have different parts that contrast such as a foreground against a background. Another element that has to be taken into consideration is the presence of different color channels.

Our analysis has been done on the USC-SIPI image database [11] and with larger bmp files found on the internet but also with images in various formats converted to bmp using imagemagick's convert tool. [12]

1.5.4 Channels and multi-byte encodings

Images and Sound are often encoded using different channels and/or multiple bytes per value per channel. Figure 6 show how the three channels in a RGB encoded image form three distinct waves representing the RG, GB and BR bigrams that can be found in the file.

When multiple bytes are used for a given channel in a file the analysis can be less obvious. When considering a file format where sound is encoded using 16 bit instead of 8 bit,

bigrams no longer represent two data points because each data point is now encoded on two bytes in the file. This means the bigrams are either the combinations of the first and second byte of one value or the second byte of one value and the first of the next one. Since the values are signed integers centered around zero this is creating the pattern seen in Figure 7 which is the same song as in 5 but encoded on 16bit.

Looking at trigrams in a 16 bit files results in even more surprising patterns. Figure 8 shows the tridimensional representation of the trigrams in the speech shown in Figure 4 but encoded on 16 bits instead of 8. Because of the low range of human speech most values will be close to zero. Which means high order bytes will be very close to 0x00 or 0xFF, the trigrams represent either the combinations of first, second and first bytes or the second, first and second bytes. This creates the projections on the two sides of the cube and on the edges where two of the three bytes are both 0x00 or 0xFF.

1.5.5 Bytecode

Patterns produced by executable code are particularly interesting because they are very useful when doing reverse engineering. Figures 9, 10 and 11 show bytecode for different architectures. While the three samples come from very different architectures we can still see similarities in the bigrams. This holds true for a large number of tested standard architectures. This is a very useful property since we can differentiate known architecture while still retaining the ability of identifying an unknown architecture as bytecode. We analysed glibc and busybox as it is easy to find those binaries compiled for a wide variety of architectures and have confirmed that the patterns remain consistent on other binaries as well. In particular we verified that they are recognizable even for smaller samples such as large shellcodes or small virii.

The vertical and horizontal lines are typical

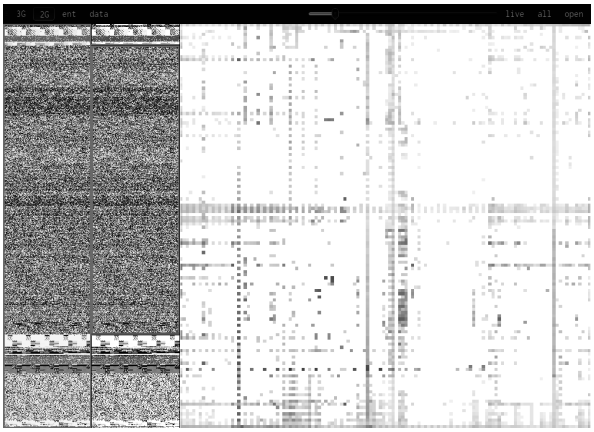

Figure 9: i386 bytecode bigrams

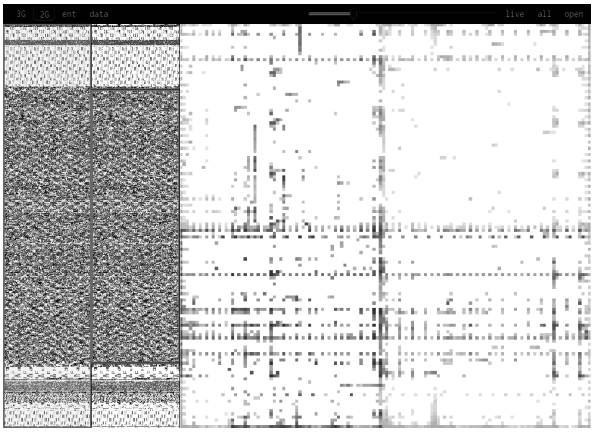

Figure 10: 64 bit powerpc bytecode bigrams

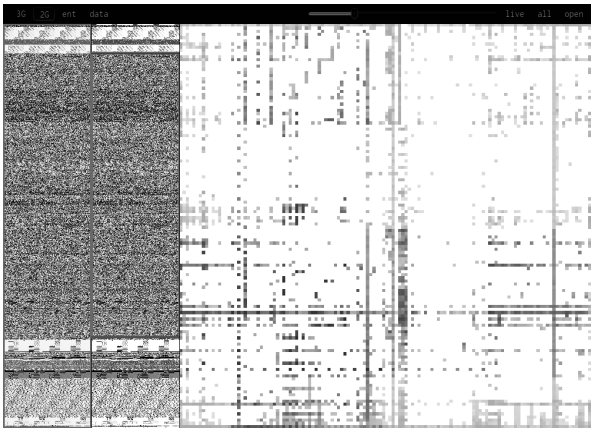

Figure 11: x86_64 bytecode bigrams

of instructions taking relative arguments and multi-byte addresses in general. The pattern is produced by the instructions or the high order bytes of addresses remaining the same while lower order bytes and instruction arguments change. This property is common to almost all binary byte codes.

1.5.6 Compressed Data

While it was expected for raw data such as images or bytecode to produce some sort of pattern the same can not be said about compressed data. Figures 12, 13 and 14 all represent compressed data but with different algorithms. Not only can we see they don't look the same, but they also seem to each have a clear signature pattern. Experiments show that this holds true for multiple files compressed with the same algorithms just as it does for bytecode. Figure 12 is a gif file, 13 a jpeg and 14 is png which uses the deflate algorithm also used by gzip. We observed the same diagonal lines in png files as shown for png in other files that where gzip compressed. This highlights the advantage of this type of analysis which effectively shows information about the data without being biased by file format headers.

Those representations also allow us to make assumptions about the efficiency of those compression algorithms. The gif from Figure 12 seems to be the worst of the three because it doesn't use as many different bigrams as the other two. Note that this also depends on the settings used while compressing the image, this is simply a side note on the usefulness of a bigram representation for visually analyzing compression rates.

1.5.7 Pseudo-random Data

Randomness analysis wasn't the focus of this research. binglide applies some filters to the data it output in order to produce better visu-

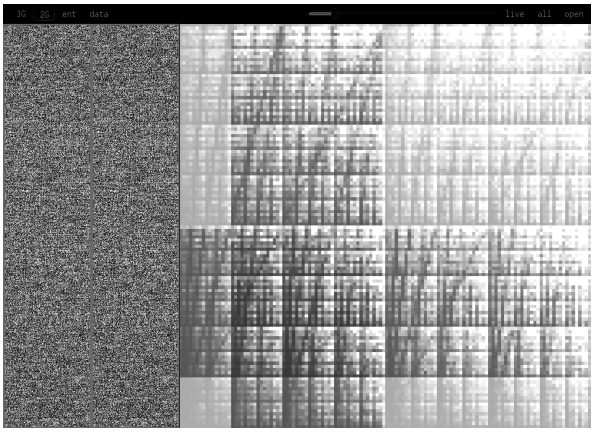

Figure 12: gif compressed image bigrams

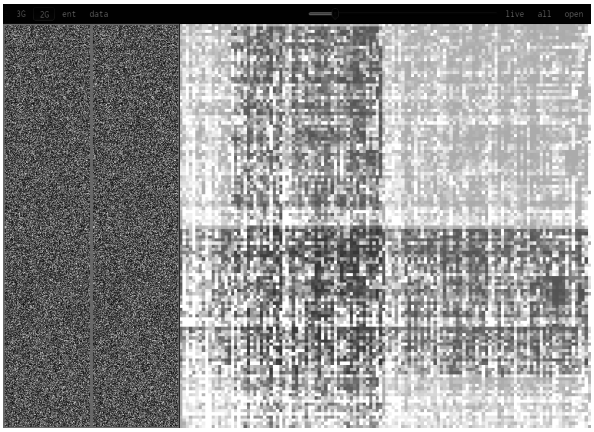

Figure 13: jpeg compressed image bigrams

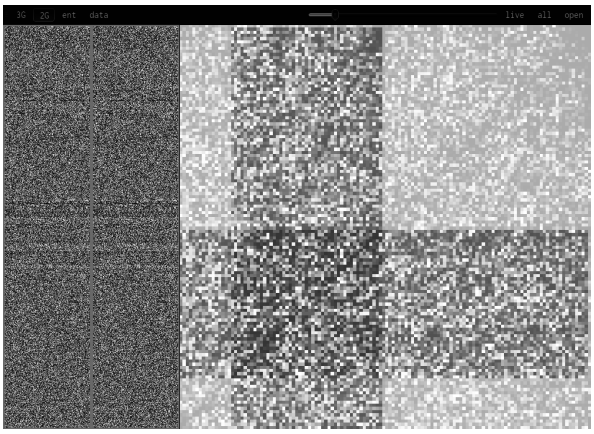

Figure 14: png compressed image bigrams

alizations in the majority of cases. When looking at bigrams of good random data instead of seeing uniform distribution of bigrams as would be expected some noise can be noticed. This is due to the fact that the most occurring bigrams are highlighted and the less occurring bigrams hidden, even if there is a very small difference. Because of this good or bad pseudo random number generators can't be analyzed with this version of binglide. This is considered for future work. Analysis of encrypted data or steganography encounters similar problems.

1.5.8 Case study: Unknown Firmware

To illustrate how binglide can be used for reverse engineering we will describe how it helps with getting a very high level overview of the layout of a file. We will look at a raw dump of the firmware of Netgear's GS108Ev2.

When looking at the whole file, which can be seen on the left pane in Figure 15, we immediately notice it is composed of four sections, on this figure we look in more detail at the first one. We see it is composed of two parts with a small padding in between. The bigram analysis of the second part on this figure tells us it is some sort of byte code, however the architecture isn't immediately obvious.

Scanning through the first part of that same section using the bigram view we notice two distinctive patterns in Figures 16 and 17, those lines probably indicate multi byte integers. We find those two locations producing two similar patterns in the data view on Figure 18. Those are almost certainly two arrays with offsets into the section or file. Once located it is only a matter of more careful examination in order to find the size of each entry and begin to search what they are used for.

When looking at the other sections in the file we find that the exact same patterns are repeated for each one. This illustrates how visual reverse engineering can very quickly give

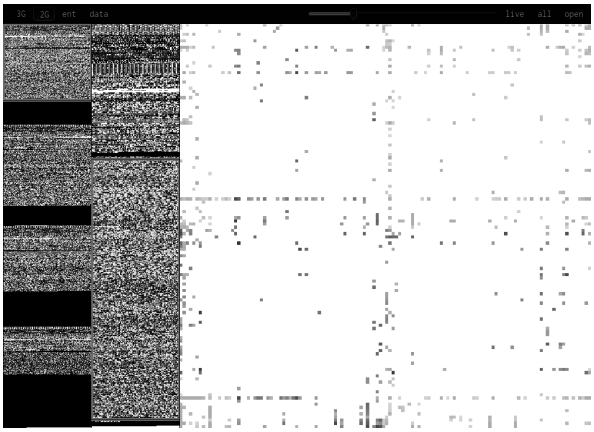

Figure 15: First section bigrams

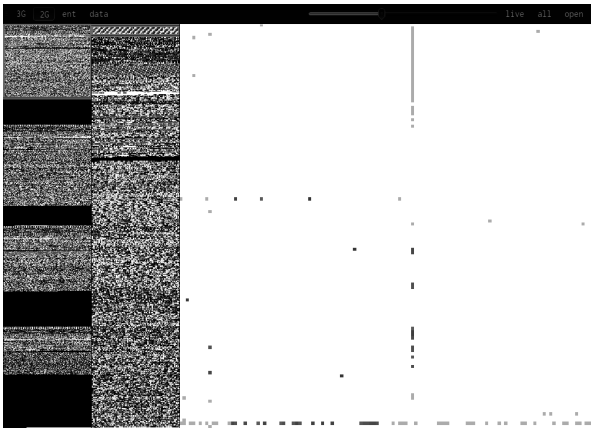

Figure 16: First array in header bigrams

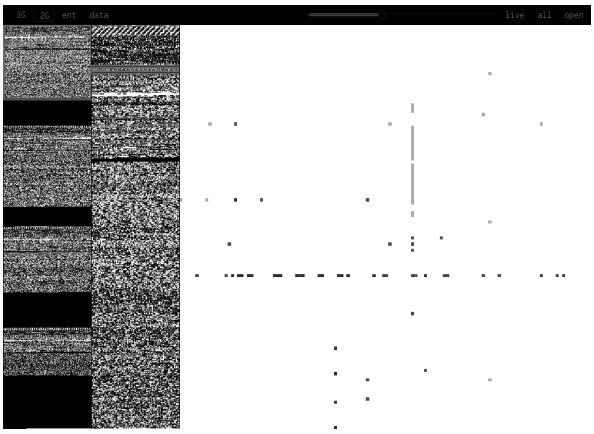

Figure 17: Second array in header bigrams

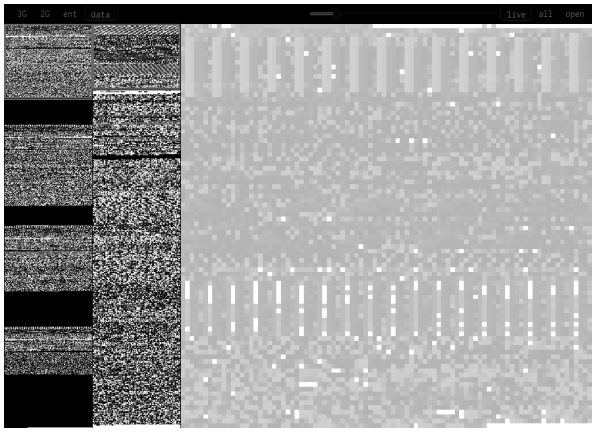

Figure 18: Data overview of header

insight into unknown data formats and provide a starting point for further analysis. Doing this work exclusively with text based tools would take significantly more time and effort.

1.6 Reception

binglide has been released on github [2] where it received generaly postive reviews. A few days after release the project got over 100 stars. People are still trying out the project and it is to soon to tell if it will be considered useful. The feedback we did get was very positive and even asked for integration with software such as the Radare2 framework or IDA Pro. Another very application specific feature that was asked for is new algorithms tailored at identifying which modulations where being used in unknown radio transmissions, mainly for use with software defined radio related work.

We expected some concern regarding performance because the tool was released in a proof of concept stage but this was an issue only for a small minority of users as most users seemed to use it on relatively small files (tens of megabytes).

The release did create start a dialog with the community, mainly through twitter and github, which resulted in very interesting sugestions for technologies to use for the next version and some people offering to help with development.

1.7 Other Work

1.7.1 Sonification

My supervisor, Dr Julio Hernandez-Castro, sugested using sonification as an alternative to visualisation. Different ways of doing this where tried and one was found to be very effective.

Two solutions require the space representing bigrams or trigrams to be mapped to a linear space in order to be projected on time or volume. In order to achieve this in a way that allows a user to effectively interpret the result we used a n-dimensional hilbert curve to enumerate the ngrams.

With this mapping it is possiblie to produce a sound sample which travels trough the space of ngrams and modulates tonality or volume according to the value of the ngram at that point. A variant of this is inspired by the way we compute the values to be rendered for visual representation, see section 1.4. We can sort the ngrams according to their value using this as a timeline and modulating the tonality or volume according to the position of the ngram on the hilbert curve.

The problem with those two sonifications is that they require a significant time to be played since we need to travel trough all ngrams and output a sound for each one. This takes between 20 and 40 seconds for each sound to be distinguishable enough to get an idea of the data type.

The reason this is so inefective is that we only output one frequency at a time while we can distinguish many more. An alternative solution uses 256 different channels $C_0...C_{256}$ each with their own distinct frequency $F_0...F_{256}$. We output 256 sounds $S_0...S_{256}$ instead of 256^n with the previous solutions, for each one we modu-

late each channel so that $F_i = V(i, j)$ where V is the value of that bigram.

Using this method we can recognize data types with audio samples as short as one or two seconds.

We showed sonification of data types works, however we haven't found a good use for it yet since visualisation is so effective and interactive. Maybe there could be some uses for example in projects offering monitoring through real time ambiant noise [13].

1.7.2 Callgraphs

Cantor Dust claims to have a very useful feature for reverse engineering which is automatic architecture-agnostic callgraph generation. The details of how this works aren't publicly available but some ideas can be tested.

In particular we tried considering all groups of bytes in a file as offsets to other parts of the file and building histograms based on those references hopping some locations would stand out. We considered each group of 2, 4 and 8 bytes as a little or big endian absolute or relative offset into the file.

At first our results where not promissing, but after eliminating locations containing printable characters we started having some correct results on small test files using the X86_64 architecture. On larger files there is to much noise for this naive implementation to have any success.

We plan on looking further into this type of algorithms in the future.

1.8 Future Work

Future work on binglide will focus on using less RAM in order to be able to analyse larger files. Also visualising streamed data is considered because it could be useful in monitoring.

Automatic detection of data types in a file is also a priority because it will enable the use of binglide for automated analysis. Another important aspect of this is the integration of binglide in other tools and frameworks such as Radare2 and IDA Pro through a plugin API.

1.9 Conclusions

The tool that was developed during this research is a proof of concept that already implements some of the most useful algorithms and will continue to evolve. It is at the moment the only tool implementing an interactive user interface while being at the same time open source, multi-platform and already released to the community.

The field of visual reverse engineering is still in its infancy and is starting to attract interest from many in the reverse engineering community [14, 15]. It is very interesting work and contributing something practical that can be actively used is very rewarding especially when met with a positive response. This work provided the author with insight into the tools that can help reverse engineers and where the state of the art might be headed.

2 Breaking the Dracon CAPTCHA

2.1 Introduction

Another part of my research has been about Flash CAPTCHAs. In particular Dracon [16] has been studied and broken in several different ways. Three attacks are presented with elements that are generic enough to limit the usefulness of implementing client-side interactive CAPTCHAs. The remainder of this section will present the Dracon CAPTCHA. Section 2.2 will present a design flaw in five out of six variants of the Dracon CAPTCHA, showing that a simple screenshot reduces this CAPTCHA to a standard image based CAPTCHA that is easy to solve using

Figure 19: Typical Dracon CAPTCHA

OCR. Section 2.3 exposes some flash-specific techniques that allow us to weaken the remaining CAPTCHA in order to break it with the same technique. Finally section 2.4 presents limits to what can be achieved regarding interaction when using a simple challenge/response model, effectively extracting cryptographic keys and bypassing the flash application by submitting challenge responses directly.

2.1.1 The Dracon CAPTCHA

The Dracon CAPTCHA has 6 variants. All of them composed of blurred letters that are revealed one after the other but never showing two at the same time. In some cases user interaction is required and one has to move the mouse pointer over a letter to unblur it. The six versions are combinations of different colors, possible and/or required mouse interaction and more or less noise in the background. Most variants also have animated rain in the foreground as can be seen in Figure 19.

The way this CAPTCHA is implemented is by having a flash media object that is downloaded by a user's browser when visiting a web page requiring them to enter a CAPTCHA. On the server side the Dracon CAPTCHA is implemented in PHP, the web page also contains opaque parameters that are given as arguments to the flash object. The code entered by the user is sent to the server with one of the opaque arguments. On the server side those two values are then given to a function which verifies if the value is correct or not. We give more insight on this in section 2.4 but during the first attacks we do not try to exploit weaknesses in the overall design.

2.2 Frame 1 Attack

The first thing needed to break a flash CAPTCHA are tools that can be automated and don't require the browser. The compiled flash file is downloaded and launched from the command line using gnash [17]. This has an added benefit: gnash doesn't seem to have implemented the functions used by Dracon to blur the letters. It is important to note that if this had been the case it could have been easily disabled by changing gnash's source code because it is open source or by using one of the techniques presented in section 2.3.

Because the letters are no longer blurred the only protection remaining in place is the animated rain on the foreground and the noise in the background. The rain is animated to fall downwards, the problem is on the very first frame it hasn't started to fall yet. This can be seen very well by taking a screenshot of the very first frame which gnash allows us to do by specifying an option on the command line. By forcing the window size to be big enough we can see the rain floating above the CAPTCHA on the first frame shown in Figure 20.

Applying some simple filters using imagemagick's convert tool [12] from the command line is enough to clean up the image enough to be broken by OCR as can be seen in 21.

This is the command used for cleaning up the image:

```
convert screenshot.png
  +dither -negate -colors 2
  -colorspace gray
  -threshold 90%
  ocr_input.jpg
```

By taking a simple screenshot and applying some basic filters we reduced the interac-

Figure 20: Dracon CAPTCHA Frame 1

Figure 21: Cleaned version of Frame 1

tive flash based CAPTCHA to an exceptionally weak standard image based CAPTCHA. This worked for five out of the six variants of the Dracon CAPTCHA. The version it didn't work on was the one labeled "Anti timed-OCR" this is because in that particular version characters never appear at the same time, blurred or not, instead they are blurry spots that move to the center one by one in a random order. They are replaced by an actual character and unblurred only when they arrive at the center. The designers clearly intended this as a protection against taking a screenshot at different time intervals in order to get a shot of each letter and then joining them. We show that isn't needed to break the other variants but offered some protection against the very simple use of gnash's inability to render blurred letters.

It is important for an animated CAPTCHA that all frames have the same difficulty, as shown in this attack one weak frame is enough to compromise the whole CAPTCHA. Randomizing the order in which letters are revealed is also important to defeat timed OCR attacks. Still even with those precautions taking the mean of all frames can be effective at defeating some CAPTCHAs. The Anti timed-OCR version of Dracon takes care of this because the letters move around and are only revealed in the same central position.

2.3 Patching the Binary

In order to break the remaining CAPTCHA we started decompiling the compiled flash file using showmycode [18] a free webservice. Standalone decompilers are also available but this was enough to get a quick understanding of how the CAPTCHA worked. The code isn't obfuscated in any way and the decompiler had no trouble showing us something we could work with. Obfuscation would not have been a solution but only something that would have required some time and effort to get around. Once we understood the different code loops and how the animation worked we looked at the structure of a swf file. Once uncompressed we where able to analyze the file using a hex editor in order to locate strings, integer constants, symbols and byte code. It is important to note that the swf downloaded from the server is always the same for a given variant of the Dracon CAPTCHA, therefor we have only 6 different swf files that are very similar for most of them. We present three types of modifications that can be made to the file and can be used to significantly weaken a flash CAPTCHA, in practice only the two first where needed to break the Anti OCR version of Dracon.

2.3.1 Disabling symbols

When using functions from the standard library in flash, the names of these functions are stored in a symbol table in the file. By corrupt-

ing the names of some symbols we can prevent them from ever being called. In a language such as C corrupting symbol tables would result in errors during the initialization of shared libraries at run-time but in flash this seems to be silently ignored and default values are used for symbols, including default return values for unknown functions. Using this we can make any particular function return some default value. The same symbol table is also used for symbols defined in the flash file itself.

2.3.2 Value patching

Constants used in the code are present somewhere in the file and therefor nothing prevents us from changing them if we can locate them. This is harder to do than locating the symbol but is necessary if we don't want the value to be 0 or if the value is a simple constant and isn't stored in a variable.

The simplest technique that doesn't require any specific tools is searching for the original value. Flash uses 32 bit signed integers stored using big-endian. If there is more than one match but not too many, an effective method is trial and error. Changing all values one by one and observing which change causes the desired behavior when the animation is run.

2.3.3 Code patching

This is most difficult but also the most powerful and is certainly an option. In theory we can completely rewrite any part of the file we need to in order for it to present the CAPTCHA in a way we can break. It is possible to make it print some information to the debug output which will be printed in the console thus giving us valuable information. This is the most telling example of the fact that the code runs on our machine and should therefor not be trusted. It is important to take this into consideration when designing interactive CAPTCHAs in Flash or other languages such as javascript.

2.3.4 Defeating Anti timing-OCR

Using two of these techniques we successfully reduced the Anti timing-OCR version of the Dracon CAPTCHA to something the attack presented in section 2.2 could handle. We where also able to weaken the CAPTCHA even more by disabling the rain completely and perfectly aligning all the letters which resulted in even better results with OCR.

The first patch we applied changed all occurrences of "rain" to "xxxx", this affected symbols such as rainAmount, rainSpeed, rainBackground, rainDropWidth, and many others. With this patch the rain was completely disabled. A full list of affected variables can be found by running the strings command on a decompressed version of the CAPTCHA and looking for "rain".

Next we patched the values of the blurInSpeed and blurOutSpeed, these control how fast each letter is blurred and unblurred. The default for blurInSpeed is 5, we set it to 0 so that the letter never actually gets blurred back after it is unblurred. The blurOutSpeed is set to something very high causing the letters to get unblurred immediately. With this patch we canceled the blur. This shows a generic way of doing this without relying on gnash as we did with the previous attack.

The protections bypassed in the two patches so far had already been shown to be ineffective in section 2.2. What is left is the centering of each letter before revealing it. In order to patch this we needed to look at the disassembled code. We found that the routine responsible for centering each letter was called centerCode. It moves the letter in the desired direction using a fixed step horizontally and verticaly. This makes for four distinct cases: up and down combined with left or right. Patching the four steps to be 0 for left and right move-

Figure 22: Patched Anti timing-OCR

ment and $+1, -1$ for up and down resulted in the centerCode routine perfectly aligning the letters for us.

Still it takes some time for this function to center the letters verticaly therefor we also patched the initial positions which is calculated in the following manner: $y = (Math.sssdom() * 15) + 5$. Setting 15 and 5 to 0 resulted in the letters being centered from the start.

In the original CAPTCHA when one letter starts to be blurred the next letter starts to be unblurred. Because those two actions are immediate in our patched version the two letters appear at the same time during the time it should have taken to blur and unblurr them. In fact all the letters of the CAPTCHA appear at the same time during the first frame because of this. As can be seen in Figure 22 this results in a very easy to break CAPTCHA.

With this attack we showed that the Dracon CAPTCHA relies to much on the code running on the client's machine while it shouldn't be trusted. We didn't even have to rewrite parts of the code. Searching for key values and changing them was enough to significantly weaken the CAPTCHA and make it very easy to break using OCR. By rewriting the code we could make it output the offsets of the sprites it chooses to display for the letters and effectively leak the displayed text directly to the debug output. There is nothing the authors could have done to prevent this. If the CAPTCHA knows what the code is we can leak it. This attack vector is explored in section 2.4 but without the need to modify the code.

2.4 Leaking the Key

The swf file that is downloaded and contains the flash object that displays the image never changes for a give variant of the Dracon CAPTCHA on a given website. This means the code that should be displayed has to be passed in the argument that is given to the CAPTCHA. It is easy to understand how it works because it is explained on Dracon's website and is easy to find in the server side PHP source code. The secEncCode argument is the text that should be displayed encrypted using AES256, the IV is constant and the key has to be contained in the flash object if it can decrypt the text which has to be the case. Looking at the disassembly we can indeed see a variable aesKey containing the key. It turns out we don't need the disassembly, searching for a string matching the regular expression "([a-z0-9]24)" is enough to extract the key from the decompressed version of any of the variants of the Dracon CAPTCHA. This key needs to be extracted only once for a given website. It can then be used to break any of Dracon's variants only by decrypting the argument given to the CAPTCHA which can be found in the web page.

Even if the CAPTCHA had been obfuscated or if the key wasn't passed as an argument but instead a different file was generated for each challenge the text could have been recovered from memory using memory dumping or a debugger.

This attack perfectly illustrates the fact that if the CAPTCHA needs to know the text it is displaying, and if that CAPTCHA runs on the client's machine then a malicious user can extract the text directly.

2.5 Limitations of flash-based CAPTCHAs

The Dracon CAPTCHA's main use of flash is to provide user interaction, however in none of the attacks we presented user interaction was a problem. In particular we never needed to emulate the use of a mouse. In fact we didn't even need to bypass it as it didn't hinder us. The remaining used feature of Flash is its ability to render animations but this can be achieved using other formats such as gif which would be less vulnerable to reverse engineering and patching.

The biggest problem with the Dracon CAPTCHA however is that it runs trusted code on the client side which makes it vulnerable regardless of any other design problems that it might have.

In general the only valid use of flash for interactive CAPTCHA's would be as an entrusted client in an implementation resembling online video games. In this setup the server would check each of the client's actions to make sure they are legal. The CAPTCHA would be presented as some sort of game where the user would have to solve a task. Many online games still suffer from the massive use of automated programs for cheating so this doesn't seem like a very robust solution either.

Closing words

This work has been done as part of my M.Sc in Computer Security under the supervision of Dr. Julio Hernandez-Castro, whom I would like to thank for introducing me to the field of Visual Reverse Engineering and for his feedback on my work.

References

[1] Christopher Domas. *Cantor Dust*. Jan. 2012. URL: https://sites.google.com/site/xxcantorxdustxx/.

[2] Wannes Rombouts. *binglide repository*. Aug. 2014. URL: https://github.com/wapiflapi/binglide.

[3] Craig Heffner. *Binwalk*. Jan. 2013. URL: http://binwalk.org/.

[4] Gregory Conti et al. "Visual reverse engineering of binary and data files". In: *Visualization for Computer Security*. Springer, 2008, pp. 1–17.

[5] Jonathan Helfman. "Dotplot patterns: a literal look at pattern languages". In: *TAPOS* 2.1 (1996), pp. 31–41.

[6] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. "A language independent approach for detecting duplicated code". In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, pp. 109–118.

[7] Gregory Conti and Erik Dean. "Visual Forensic Analysis and Reverse Engineering of Binary Data". In: Proc. Black Hat USA, Aug. 2008. URL: https://www.blackhat.com/presentations/bh-usa-08/Conti_Dean/BH_US_08_Conti_Dean_Visual_Forensic_Analysis.pdf.

[8] Christopher Domas. "The future of RE Dynamic Binary Visualization". In: Proc. Derbycon, Sept. 2012. URL: https://www.youtube.com/watch?v=4bM3Gut1hIk.

[9] Aldo Cortesi. *Visualizing entropy in binary files*. Jan. 2012. URL: http://corte.si/posts/visualisation/entropy/index.html.

[10] Travis Oliphant. “Numba python bytecode to LLVM translator”. In: *Proceedings of the Python for Scientific Computing Conference (SciPy)*. 2012.

[11] Allan G Weber. “The USC-SIPI image database version 5”. In: *USC-SIPI Report* 315 (1997), pp. 1–24.

[12] Michael Still. *The definitive guide to ImageMagick*. Vol. 1. Springer, 2006.

[13] Aldo Cortesi. *Introducing choir.io*. Aug. 2013. URL: http://beachmonks.com/posts/intro/choir.html.

[14] John Goodall et al., eds. *VizSec '13: Proceedings of the Tenth Workshop on Visualization for Cyber Security*. Atlanta, Georgia: ACM, 2013. ISBN: 978-1-4503-2173-0.

[15] *SecViz, Security Visualization Community*. URL: http://secviz.org/.

[16] Dracon Ltd. *Visual Flash CAPTCHA*. Sept. 2012. URL: http://www.dracon.biz/captcha.php.

[17] Gnash Project. *Gnash, opensource flash player*. URL: http://www.gnashdev.org/.

[18] *Show My Code is a Free Online Decoder / Decompiler*. URL: http://www.showmycode.com.