Automated Forensic Extraction of Encryption Keys Using Behavioral Analysis

Gareth Owen
University of Greenwich
United Kingdom
g.h.owen@gre.ac.uk

***Abstract*—In this paper we describe a technique for automatic algorithm identification and information extraction from unknown binaries. We emulate the binary using PyEmu forcing complete code coverage whilst simultaneously examining its behavior. Our behavior matcher then identifies specific algorithmic behavior and extracts information. We demonstrate the use of this technique for automated extraction of encryption keys from an unseen program with no prior knowledge about its implementation. Our technique can also be used for automatic categorization and suggestion of function purpose to analysts.**

Keywords: binary analysis, encryption key extraction, behavioural analysis.

I. INTRODUCTION

Software is often written by a programmer in a high-level language which is then compiled into machine code (a binary) for execution on the system. Software is then distributed in binary form without the original high-level source code. This process of conversion (compilation) from a high-level language to a binary is a lossy process. Comments, structure naming, code layout are all lost in this conversion process as they only serve to aid computer programmers in reading the code and are not required for execution.

Binary analysis is used to discover the function of software after compilation to detect security holes or verify the software performs as intended (verification). As key information used by humans to decipher code is removed, the task of binary analysis is exceptionally time consuming. Often optimized sections of code bear little resemblance to the high level code typed by the programmer – requiring the analyst to painstakingly examine the assembly code line by line.

The binary analysis task is further obstructed because it is not trivial to recognize common code such as that in statically linked libraries. Tools such as IDA Pro [1] have made some efforts to tackle this problem by compiling a database of library function fingerprints; however, a new or obfuscated version will fail recognition. Further consider two programmers who implement the RC4 [2] encryption algorithm, both programmer's code behaves the same but they may be coded in radically different ways.

Consider now the task of trying to discover an encryption key embedded in a program or stored in a running process's memory dump. Unless the encryption algorithm *implementation* is previously known it is a slow and painful task of binary analysis to identify where the key is stored.

In this paper, we set out a method of recognizing code based on its behaviour and apply it to a number of different implementations of the RC4 encryption. Instead of examining the compiled instructions and trying to perform a pattern match, we examine the behaviour, the net effect of the instructions and pattern match against this. This enables matching regardless of the particular implementation choices or style of the programmer and also allows matches where code has been obfuscated. We then go on to show that it is possible to automatically recover the key without prior knowledge of the particular implementation.

We envisage that this technique will not only be used by binary analysts to extract keys from static binaries but also by forensics analysts to extract encryption keys from process memory dumps acquired through a cold memory attack [3]. Forensics is often hindered or completely obstructed by the use of cryptography but the approach presented in this paper paves the way for automatic extraction of encryption keys from a system/process memory dump.

II. RELATED WORK

Anti-virus software uses signatures as a tool in its arsenal to recognise known viruses by storing stub instruction combinations; however, this is fraught with difficulty [4] as viruses often 'evolve' or use complicated polymorphic obfuscation techniques [5]. Therefore, anti-virus has evolved to examine behaviour of viruses through monitoring API calls. Wagener [6] proposes use of a phylogenetic tree to identify behaviours of similar malware based on the system and API calls they make. Identifying malware by its API call behaviour is a common technique but using the same technique to detect algorithmic behaviour is fraught with difficulty.

Rhee [7] proposes profiling the data object access behaviour of malware and using this information to identify it. He applies the technique to detecting rootkits and claims no false positives on uncompromised kernels. Their work has some small similarities with ours to the extent that we do not

examine API calls, except our focus is identifying specific algorithmic behaviour and then extracting encryption keys whereas they focus on producing better malware signatures.

There has been a substantial body of work on identifying and automatically unpacking malware regardless of the packing technique used. Omniunpack [8] and PolyUnpack [9] assume that packers have fairly similar behaviour characterised by, for example, execution of a dynamicly generated memory page allowing identification of the unpacker and entry point. Lyda [10] proposes identifying packed executables based on the level of entropy in their code section. Code sections typically have low entropy whereas packed code often appears to be random. This technique could be used to identify code which changes the entropy of memory, allowing for the potential identification of code sections that employ cryptography; however, manual analysis would still be required for specific algorithm determination and key extraction.

Our work is similar to that on detecting and automatically extracting packed malware to the extent that we exmaine non-API call behaviors; however, our approach is generalizable to a wide spectrum of algorithms and behaviours located anywhere within the binary.

III. BINARYMO - BEHAVIOURAL ANALYSIS FRAMEWORK

In this section we present BinaryMO, our binary behaviour analyzer. We describe the techniques used for recognizing functionality in an executable without consideration to the specific implementation choices. This technique is able to identify algorithms, such as encryption, regardless of the finer details of their implementation.

BinaryMO emulates a program and analyzes the behaviour of the code rather than the particular way in which it was implemented. We then use a set of algorithms to detect particular behaviours and then extract any required information. For emulation we use the PyEmu x86-32 emulator for Python [11]. PyEmu provides an emulated CPU, paged memory and limited system call emulation. PyEmu also provides us with the ability to step through binaries one instruction at a time hooking memory access, register access, individual instructions, etc. PyEmu's CPU emulator is mostly complete but omits a number of instructions used in some binaries we analysed (namely 8-bit operations) – we have therefore implemented these instructions and will contribute our additions freely. But, we caution other researchers that some of the instructions omitted do not cause PyEmu to throw an exception but are silently ignored.

If one starts with the executable to be analysed, the executable file can be broken into two main sections: data and code. Each section must be loaded into memory at the specified address and the executable must be linked with the desired libraries. Alternatively, if we are using a forensic memory dump then this will be loaded verbatim into memory and the process context used to initialise registers and the instruction pointer. The code section is then scanned for call instructions and the call address noted in a symbol table. This provides us with a list of functions within the program and their respective offset. We then emulate the program aiming for complete code coverage whilst allowing a set of behaviour modules to match desired traits. As soon as complete coverage is achieved, the program terminates.

If code coverage is incomplete and the program enters a large loop or exits, it may become necessary to intervene. At this stage, we build a tree of function calls in the binary with the tree head as the entry point. We then build a list of the unexecuted functions ordered by their depth in the tree (ignoring children of unexecuted functions). We examine the references to each function and attempt to extract static parameters and dynamic values that might have been determined during prior emulation. Where this fails to determine a parameter, a dummy value is inserted after type determination. Each function is then emulated in turn as illustrated in Figure 1 and the process is repeated if code coverage is still incomplete.

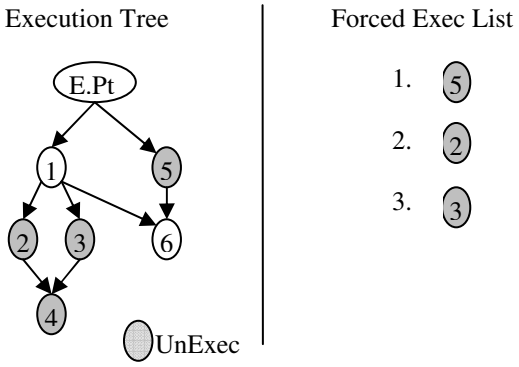

Figure 1: Forced execution determination

Next, the analyzer will load a library of code matching modules for the particular functionality the analyst is looking for. Each module will have three components:

Trigger: The purpose of the trigger is to identify code that MAY be what this module is looking for. It will often look at the initial behaviour from an algorithm and trigger the confirmatory stage. The reason for separating the trigger and confirmation is one of overhead – emulation is slow and overly onerous matching will slow down performance substantially. The trigger acts as flag, flagging up interesting code and encouraging analysts to produce simple triggers.

Confirmation: The confirmation section will confirm that the code is that which we are looking for. At this stage, it can either identify the code to the analyst or if it is likely the analyst will desire particular information from the code (such as keys) then it will move to the extraction stage.

Extraction: If called by the confirmatory stage then this part will either continue execution to acquire the relevant information or examine the emulation log to extract it.

Figure 2 gives an overview of how the components of the behavioural analyzer fit together.

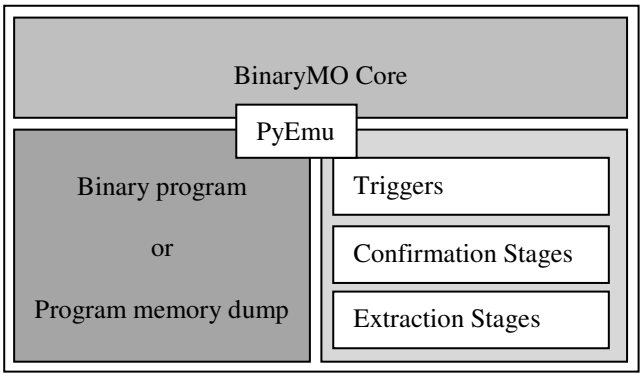

Figure 2: Behavioural analyzer overview

There are evidently some important points to this approach that require consideration, as follows:

1. **Code coverage**: As we force code coverage when it is not achieved readily we inevitably put some functions into unexpected states; analysts should bear this in mind when developing matchers. Ideally, we will be matching *algorithmic* behaviour rather than tracking stack variable contents.
2. **Code linearity**: When an algorithm is divided across a number of functions, the analyzer depends on the functions being executed in sequence to provide a positive match. This is likely to happen in all cases but is wholly dependent on the intelligence of the code coverage functionality. We believe the forced execution tree described above mitigates this problem as best as possible.
3. **Library calls**: It is often undesirable to execute library calls particularly from malicious code. Where possible, the call is either emulated or replaced with a stub. The latter has the effect of changing the behaviour of an algorithm if it is dependent on library call and therefore we discourage the use of this approach where this is the case.

In the next section, we provide an example of matching the RC4 algorithm with high probability and extracting the encryption key.

IV. BINARYMO: AUTOMATED RC4 KEY EXTRACTION

We apply the behavioural analysis technique to the RC4 algorithm [2] to automatically extract the encryption key. We have chosen RC4 because of its simplicity for the purposes of demonstration; however, the behavioural technique is applicable to any other cipher.

The RC4 algorithm consists of two parts: A key scheduling algorithm whose purpose is to turn an encryption key, K, into a randomly ordered initial permutation S of $\{0, \dots, N-1\}$, and a key stream output part which uses this permutation to generate the pseudo-random key stream sequence.

The algorithm first initializes S to the ordered set $\{0, \dots, N-1\}$, and then initializes i and j to zero. The algorithm loops N times incrementing i as a counter, and incrementing j pseudo-randomly based upon the key. For each iteration, $S[i]$ and $S[j]$ are swapped resulting in a pseudo-random shuffle of S based upon the key. The algorithm then goes on to enter a second loop, iterating once for each key stream byte. In each iteration, the algorithm selects an entry from S pseudo-randomly and performs one additional swap.

The algorithm is set out in Figure 4 with all variables modulo N.

Key Scheduling Algorithm (KSA)	**Pseudo Random Generation Algorithm (PRGA)**
Initialization	**Initialization**
$S = \{0, \dots, N-1\}$	$i = j = 0$
$i = j = 0$	
Key-based shuffle	**Output Loop**
For $i = 0 \dots N-1$	$i = i + 1$
$j = j + S[i]$	$j = j + S[i]$
$+ K[i \bmod K_{len}]$	Swap $S[i], S[j]$
Swap $S[i], S[j]$	Output $S[S[i] + S[j]]$

Figure 3: RC4 Stream Cipher

Our goal is to automatically identify either K or $\{S, i, j\}$ after recognising the algorithm in the binary. K is the Holy Grail, with which we should be able to decrypt all data; $\{S, i, j\}$ would allow us to decrypt future data and make it substantially easier to work backward (vs. no knowledge). Strictly a program does not require K after initialisation of RC4 but it is rarely discarded because typical implementations frequently re-initialised RC4 with a different initialization vector ($K + IV$). For example, the Wireless Equivalence Protocol would both change the initialization key of RC4 frequently based on the original key.

We first need to identify particular behaviours which are characteristic of RC4 and not easily obfuscated by programmers or compiler optimisation. The code in Figure 3 is a fairly typical implementation and is already optimised. The initialization of S to the ordered set $\{0, \dots, N-1\}$ is a prime target for detecting the start of RC4 (a trigger); however, this could be obfuscated by initializing S in some other order than linearly ascending; however, regardless the

behaviour is a series of $\log N/\log 2$ sized writes in a block S of size N where the write value compiles with the following (after all writes complete):

$$\forall S[i] \in S, S[i+1] = S[i] + 1: \quad (1)$$

Therefore, we select this as our trigger – the first part of the initialisation of *S*. As *N* is almost always 256 it makes it trivial to write a memory write hook to detect this behaviour. Whilst processor architecture may dictate a different write size, the net effect of each write will always be a single byte change in memory and so our hook should look for behaviour where the net effect is a single byte write.

Once the trigger has identified the code as a potential RC4 algorithm, we enter the next stage, *confirmation* and *key extraction*. To confirm this code is indeed RC4, we hook memory and look for a series of 2*N* read and writes in the same *N* sized block as the trigger. By limiting our hook to the *N* sized block, we eliminate the appearance of read/writes to temporary variables that may be used. Once confirmed, we compare the values of the read and writes – the read and write values must perform a swap inside *S*. Although the order may vary, the sequence is likely to appear as two reads followed by two writes as follows:

$$\begin{aligned} x &= S[i] \text{ READ} \\ y &= S[j] \text{ READ} \\ S[i] &= y \text{ WRITE} \\ S[j] &= x \text{ WRITE} \end{aligned} \quad (2)$$

If one then extracts the offset into *S* for each read and write, we will see that there will be two sets of numbers, where the value of the first set increments linearly to *N – 1* and the other set increments pseudo-randomly. These offsets represent *i* and *j* respectively. This gives a confirmation of RC4 KSA completion. Knowledge of both *i* and *j* for the execution of this KSA shuffle allows us to calculate the key. For each series of two reads and two writes, where j_i denotes the value of *j* at iteration *i*, one calculates the key as follows (all variables modulo *N*) using the value of *S* before the swap writes:

$$K[0] = j_0 \quad (3)$$

$$K[i] = j_i - j_{i-1} - S[i] \quad (4)$$

Given the set *K,* one can then determine the key length, K_{len}, by identifying the repeating sequence length and then extract the key $K[0 \ldots K_{len} - 1]$.

An alternative approach would be to locate the key in memory or a processor register, but given that there are several places the key could be stored and then subsequently manipulated we would need to implement recognition algorithms for each. By calculating the key from *i* and *j* during the KSA shuffle our technique is not concerned with the location of the key but determined from the behaviour of the code – essentially abstracting us another layer from the implementation choices of the programmer.

V. EVALUATION

To evaluate our approach for key extraction, we took a variety of RC4 implementations. The implementation in Figure 4 below was taken from the Government Communications Headquarters (GCHQ) Cyber Security challenge [12]: a binary analysis challenge set by one of Britain's intelligence agencies which uses RC4 to obfuscate data in the program.

	x86 assembly	**Comments**
1	ksainit:	
2	mov [esp+ecx],cl	S[i] = i
3	inc cl	i++
4	jnz ksa_part1_loop	
5		
6	xor eax,eax	
7		
8	mov edx,0xdeadbeef	The key
9		
10	ksa_key_loop:	
11	add al,[esp+ecx];	j += S[i]
12	add al,dl	j += K[i mod len]
13	ror edx,0x8	Next byte from key
14	mov bl,[esp+ecx]	Read S[i]
15	mov bh,[esp+eax]	Read S[j]
16	mov [esp+eax],bl	Write S[j]
17	mov [esp+ecx],bh	Write S[i]
18	inc cl	i++
19	jnz ksa_key_loop	

Figure 4: RC4 assembly code

The profile of this code in terms of read and writes is that the *ksainit* function performs the initial *S* fill exactly as described in the last section and correctly triggers the confirmation and extraction stage. The *ksa_key_loop* function performs the KSA shuffle; however, it performs an extra read on line 11 but because our behaviour analyser is configured to ignore identical sequential read/writes forcing conformation with behaviour profile.

In Figure 5, we illustrate the actions of the memory hook once the extraction stage has begun. The analyser splits the sequence of reads and writes into blocks representing the iterations of the KSA shuffle and then extracts *i* and *j*. The calculations outlined earlier are then used to extract the 0xDEADBEEF key as follows:

Action	**Offset**	**Value**
Read	00h	00h
Read	00h	00h
Read	EFh	EFh
Write	EFh	00h
Write	00h	EFh
$i = 0$		

$j = EFh = 0 + 0 + K[0]$ $K[0] = \boldsymbol{EF}h$		
Read	01h	01h
Read	01h	01h
Read	AEh	AEh
Write	AEh	01h
Write	01h	AEh
$i = 0$ $j = AEh = K[0] + S[1] + K[1]$ $K[1] = AEh - K[0] - S[1] = \boldsymbol{BE}h$		

Figure 5: Read / write memory hook

In Figure 5, both *i* and *j* are inferred from the memory offset relative to the start of *S*. As *i* always begins at zero and *j* is always based on the key it is possible to determine which read and write are responsible for each. In this example, the technique extracts EFh and BEh, the first two bytes of the key, and if allowed to run for an additional two iterations would have extracted ADh and then DEh, revealing the complete key.

As discussed earlier, the key has been extracted from the behaviour of the algorithm rather than locating it in the *edx* register. By doing this, we ensure the technique is applicable regardless of where the programmer chooses to store the key.

VI. Conclusions

We have presented a general framework for behavioural analysis of algorithmic behaviour in binaries. We have then used this approach to automatically extract the key from a variety of RC4 implementations.

There are two threads to our future work. The first is applying the approach to other cipher algorithms so it is possible to extract keys from more than just RC4. We will also investigate whether it is possible to produce a behavioural matcher for a generalised class of ciphers (e.g. feistal block ciphers). The second strand to our future work is extracting keys from memory dumps obtained through cold-boot attacks – specifically we will first reconstruct the virtual memory from the page tables and then perform an automated analysis of the program to identify the key in a similar way to outlined in this paper. We also envisage that this technique can be used to automatically profile a program and suggest to the binary analyst what the behaviour of each function may be (e.g. HTTP request, etc) and even categorise functions automatically.

VII. Bibliography

[1] Ilfak Guilfanov. (2012) IDA Pro Disassembler. [Online]. http://www.hex-rays.com/products/ida/index.shtml

[2] Scott Fluhrer, Itsik Mantin, and Adi Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," in *Selected Areas in Cryptography*.: Springer Berlin / Heidelberg, 2001, vol. 2259, pp. 1-24.

[3] Alex Halderman et al., "Lest We Remember: Cold boot attacks on encryption keys," in *Proc. 17th USENIX Security Symposium*, San Jose, CA, 2008.

[4] Adrian Stepan, "Improving proactive detection of packed malware," *Virus Bulletin*, 2006.

[5] A.H. Sung, J. Xu, P. Chavez, and S Mukkamala, "Static analyzer of vicious executables (save)," in *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference, IEEE Computer Society*, 2004, pp. 326-334.

[6] Gerard Wagener, Radu State, and Alexandre Dulaunoy, "Malware behaviour analysis," *Journal in Computer Virology*, pp. 279-287, 2008.

[7] Junghwan Rhee, Zhiqiang Lin, and Dongyan Xu, "Characterizing kernel malware behavior with kernel data access patterns," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, New York, 2011, pp. 207-216.

[8] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha, "OmniUnpack: Fast,Generic, and Safe Unpacking of Malware," in *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[9] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee, "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," in *22nd Annual Computer Security Applications Conference*, 2006, pp. 289-300.

[10] R Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40-45, 2007.

[11] Cody Pierce. (2012) PyEmu. [Online]. http://code.google.com/p/pyemu/

[12] GCHQ. (2011) GCHQ CanYouCrackIt. [Online]. http://www.canyoucrackit.co.uk