INL/CON-08-14597
PREPRINT

Finding Cryptography in Object Code

SECTOR 2008

Jason L. Wright

October 2008

The INL is a
U.S. Department of Energy
National Laboratory
operated by
Battelle Energy Alliance

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint should not be cited or reproduced without permission of the author. This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, or any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for any third party's use, or the results of such use, of any information, apparatus, product or process disclosed in this report, or represents that its use by such third party would not infringe privately owned rights. The views expressed in this paper are not necessarily those of the United States Government or the sponsoring agency.

Finding Cryptography in Object Code

Jason L. Wright
Idaho National Laboratory
jason.wright@inl.gov

September 26, 2008

Abstract

Finding and identifying cryptography is a growing concern in the malware analysis community. In this paper, a heuristic method for determining the likelihood that a given function contains a cryptographic algorithm is discussed and the results of applying this method in various environments is shown. The algorithm is based on frequency analysis of opcodes that make up each function within a binary.

1 Introduction

Finding and identifying cryptography is a growing concern in the malware analysis community. The current state of the art is to locate it manually and identify it based on various constants used by specific algorithms. The approach outlined in this research is to examine the instructions that make up a function and make a determination on how likely that function is to contain cryptography. The properties of cryptographic functions from an opcode point of view are examined and samples of *findcrypto*'s use are discussed.

This work was inspired by two related pieces of work. *findcrypt[Gui06a]/findcrypt2[Gui06b]* locate various constants used in the initialization of cryptographic algorithms and further provides identification of the specific algorithm. *findcrypto* differs from this work mainly because it looks at the instructions that make up the algorithm and not the data it uses (for initialization or otherwise).

The hack on the Mifare smartcards[NESP08] involved reverse engineering the hardware by examining the distribution of logic gates. Specifically, the authors looked for XOR gates that were strongly interconnected, but where the functional block itself was loosely connected to the rest of the chip. In other words, the authors were looking for the pipelining (strong interconnection) components of the cryptographic algorithm, and its inputs and outputs (loose coupling to the rest of the chip). This type of examination is looking for the properties of cryptographic algorithms that make then stand out from normal functionality.

The type of frequency analysis is similar in nature to [Bil07]. The focus of *findcrypto* is on cryptographic algorithm discovery versus malware prediction (i.e. given a binary we want to know whether it contains crypto, not whether it might be malware).

2 Method

The software developed for this project, *findcrypto*, attempts to detect and locate cryptographic algorithms within object code by examining the instructions that make up each function within a target binary. The instructions are used to determine the likelihood that cryptographic algorithms are present.

Each binary (library or executable), is disassembled to its constituent instructions. When working with libraries, the symbols are available and the starting and ending of each function is known. Modern disassemblers, like IDA Pro, will attempt to delineate function boundaries and will assign names for each. The symbol name information (if even available) is not used as an indicator of the type of algorithm by *findcrypto*, except for verification during development.

Instead, *findcrypto* loops over each function examining the instructions that comprise it. A weight is assigned for each instruction and the sum of all the instruction weights for each function is stored. The result of sorting the functions by total weight is a numerical indication of the amount of cryptographic algorithm-like behavior exhibited by the function. This weighting of functional characteristics allows a relatively quick analysis of a binary to determine whether cryptographic algorithms are present and pinpoints their locations for further analysis.

The primary method used in developing *findcrypto* has been empirical trial and error. It is difficult to decide on a representative binary for analysis, so work thus far has focused on the C library from OpenBSD and Linux (*glibc*). The following sections discuss some of the unique properties of cryptographic algorithms as empirically determined.

2.1 Opcode: XOR

XOR, exclusive-or, is an opcode implemented on most modern processors. It performs a bitwise exclusive-or operation on two registers, a register and a memory location, or a register and an immediate (constant). On *i386*, the XOR instruction is used to get a zero value into a register, e.g. `xor %eax, %eax`, which results in zeroing the *%eax* register.

On Linux (*i386*, *glibc* 2.6.1, *gcc* 4.1.2), XOR occurs 5752 times and of those, 5345, 93% are zeroing XOR operations (source and destination registers are identical).

Outside of cryptography, the XOR operation is not very common. The remaining 407 XOR uses are distributed among 135 different functions with 64 distributed in two functions *_des_crypt()* and *des_encrypt()* with 22 and 42, respectively.

Similarly, for OpenBSD (*i386*, *gcc* 3.3.5), XOR occurs 3536 times, of which 2033 (57%) are zeroing XOR. The remaining 1503 uses are distributed among 60 functions. Of the 60 functions, 21 are cryptographic and account for 1393 uses of non-zeroing XOR. *SHA1Transform()* alone accounts for 312 non-zeroing XOR operations.

Because of the high frequency of non-zeroing XOR in cryptographic functions, it is given a high heuristic weight.

2.2 Opcode: ROL

The Rotate Left operation, ROL, rotates the bits in a register by some number of specified bits. The specification can come from either a register or an immediate. As with XOR, ROL does not often occur outside of cryptographic algorithms. Unlike XOR, however, there is no C operator that corresponds to ROL, so the only way that ROL can be emitted is by inline assembly or compiler optimization.

On OpenBSD, ROL occurs 684 times, and of those, 677 (99%) are in cryptographic functions. *RMD160Transform()* alone accounts for 295 uses. On Linux, there are only 18 occurrence of ROL, and none of them occur in cryptographic functions. Because of the high rate in OpenBSD, ROL is given a non-zero weight for cryptography.

2.3 Opcode: ROR

Rotate Right (ROR) is the same as ROL except that it rotates the specified number of bits right (towards the least significant bit). This is another opcode found to occur frequently in cryptographic algorithms.

On Linux, it occurs 67 times in the C library, 9 times within cryptographic functions (8 in *_des_encrypt()*. On OpenBSD, ROR occurs 55 times across 31 functions, none of which are crypto. As a result, ROR is given a low heuristic weight.

2.4 No Floating Point

In cryptography, bit for bit reliability between source and destination must be guaranteed. The differing algorithms and rounding practices used in floating point implementations precludes their use in cryptography. Instead, cryptographic algorithms rely on relatively simple, bit-for-bit operations.

2.5 Other Observations

When first implemented, the algorithm defined above was combined with a density calculation. Essentially the weight of the suspicious opcodes was divided by the total number of instructions in the function. This turned out to be a bad approach because a number of functions, e.g. *fabs()* (compute the absolute value of a floating point number) have XOR operations in them, but are very small (6 instructions in the case of *fabs*. This combination means that short functions are given high density over functions that really consist of cryptography. While these outlying functions are very small, it increases the amount of chaff to be examined versus using the unbiased weight.

The opcodes listed in previous section are not all inclusive of those examined by *findcrypto*. Generally speaking the number of unique opcodes required by cryptography is fairly small and the combination of all of the opcodes found within one function is the basis of this work. Each instruction is examined to see if it provides a hint towards or against the function having crypto.

OpenBSD-2.5 gcc 2.8.1		OpenBSD-4.3 gcc 3.3.5	
Score	Function	Score	Function
4000	SHA1Transform	4230	SHA1Transform
2320	RMD160Transform	2755	RMD160Transform
2240	skipjack_forwards	2240	skipjack_forwards
2240	skipjack_backwards	2240	skipjack_backwards
896	Blowfish_decipher	1440	MD5Transform
809	cast_setkey	880	MD4Transform
801	MD5Final	811	cast_setkey
750	cast_encrypt-0x2000	532	Blowfish_encipher
548	Blowfish_encipher	532	Blowfish_decipher
501	MD4Final	494	cast_encrypt
462	cast_encrypt	494	cast_decrypt
462	cast_decrypt	433	SHA512_Transform
300	xdr_callmsg	266	SHA256_Transform
300	__aout_fdnlist	90	ntohs-0x3a
206	crypt-0xba0	90	ntohl-0x3a

Table 1: Comparing Compiler Versions

3 Analysis

Being based on heuristics, it was important to test *findcrypto* on various compilers, architectures and operating systems. This section details the findings for each. For each of the sections below, the architecture is Intel IA32 (aka i386).

3.1 Different Compiler Versions

To compare various compiler versions, the C library for various releases of OpenBSD were examined. OpenBSD was chosen because its C library contains several cryptographic algorithms as well as normal C library routines like *printf()*, *fopen()*, etc. The results are in Table 1

The various GCC versions reflect the times. Version 2.8.1 was the last of the official 2.x series followed by EGCS with then became the 3.x series. This compiler has changed optimization strategies several times between each release, but for similar functions, e.g. *SHA1Transform*, the score computed by *findcrypto* varies by less than 15 percent. The relative order stays the same in most cases, with notable exceptions being those functions added to the C library during successive OpenBSD releases, e.g. *CAST-128*.

3.2 Effect of Optimization

Table 2 details the top 15 results for different compiler optimizations on the C library of OpenBSD 4.3. Between O0 (no optimization) and O2 (most optimizations enabled), there is little difference. The relative order stays mostly the same and the score changes by less than 20 percent.

On the other hand, between O2 (the highest supported by the OpenBSD developers) and O3, several more functions related to the BLOWFISH algorithm appear in the list. The additional optimizations enabled by this level, register renaming and inline functions, cause several functions to be pulled inline, increasing the score of the function as a whole.

3.3 Different Operating Systems

For comparison, the C library on Linux (Gentoo with *glibc* version 2.6.1, compiled with *gcc* version 4.1.2) was compared with OpenBSD 4.3 (compiled with *gcc* version 3.3.5). On Linux (*glibc*), only one cryptographic function is normally found in the C library: *des_encrypt()*, which as its name implies, is an implementation of DES. As shown earlier, OpenBSD contains *arcfour*, *CAST-128*, *BLOWFISH*, *MD5*, *SHA1*, and more.

Given this base of comparison, the expected results should be that *des_encrypt()* occurs at the top of the list. Table 3 confirms this and shows the rest of the top 15 from Linux compared with the same list from OpenBSD 4.3.

In the Linux results, the numbers fall off quickly from over 500 to less than 100. This is expected given the lack of cryptographic functions in *glibc*.

OpenBSD 4.3/gcc 3.3.5					
-O0		-O2		-O3	
Score	Function	Score	Function	Score	Function
4000	SHA1Transform	4230	SHA1Transform	4230	SHA1Transform
2240	skipjack_forwards	2755	RMD160Transform	2755	RMD160Transform
2240	skipjack_backwards	2240	skipjack_forwards	2240	skipjack_forwards
2080	RMD160Transform	2240	skipjack_backwards	2240	skipjack_backwards
1120	MD5Transform	1440	MD5Transform	1440	MD5Transform
878	cast_setkey	880	MD4Transform	1120	Blowfish_expandstate
640	MD4Transform	811	cast_setkey	1108	blf_cbc_decrypt
596	Blowfish_encipher	532	Blowfish_encipher	1076	Blowfish_expand0state
596	Blowfish_decipher	532	Blowfish_decipher	880	MD4Transform
510	cast_encrypt	494	cast_encrypt	811	cast_setkey
510	cast_decrypt	494	cast_decrypt	554	blf_cbc_encrypt
436	SHA512_Transform	433	SHA512_Transform	544	blf_ecb_encrypt
269	SHA256_Transform	266	SHA256_Transform	544	blf_ecb_decrypt
96	des_do_des	90	ntohs-0x3a	532	blf_enc
90	ntohs-0x3a	90	ntohl-0x3a	532	blf_dec

Table 2: Compiler Optimizations

OpenBSD-4.3	Linux/glibc
Function (Score)	Function (Score)
SHA1Transform (4230)	des_encrypt (502)
RMD160Transform (2755)	des_crypt (312)
skipjack_forwards (2240)	__strchrnul (162)
skipjack_backwards (2240)	strchr (162)
MD5Transform (1440)	strrchr (141)
MD4Transform (880)	__memchr (122)
cast_setkey (811)	____strtold_l_internal (111)
Blowfish_encipher (532)	____strtof_l_internal (111)
Blowfish_decipher (532)	____strtod_l_internal (111)
cast_encrypt (494)	____wcstold_l_internal (97)
cast_decrypt (494)	____wcstof_l_internal (97)
SHA512_Transform (433)	____wcstod_l_internal (97)
SHA256_Transform (266)	strcat (81)
ntohs (90)	__rawmemchr (81)
ntohl (90)	_IO_vfwprintf (71)

Table 3: Comparing C Libraries/OS

OpenBSD 4.3/gcc 3.3.5	
i386	*SPARC64*
Function (Score)	Function (Score)
SHA1Transform (4230)	SHA1Transform (3120)
RMD160Transform (2755)	skipjack_forwards (2240)
skipjack_forwards (2240)	skipjack_backwards (2240)
skipjack_backwards (2240)	RMD160Transform (1280)
MD5Transform (1440)	MD5Transform (1120)
MD4Transform (880)	cast_setkey (760)
cast_setkey (811)	clnt_broadcast (730)
Blowfish_encipher (532)	getanswer (670)
Blowfish_decipher (532)	MD4Transform (640)
cast_encrypt (494)	Blowfish_encipher (500)
cast_decrypt (494)	Blowfish_decipher (500)
SHA512_Transform (433)	res_init (430)
SHA256_Transform (266)	cast_encrypt (370)
ntohs-0x3a (90)	cast_decrypt (370)
ntohl-0x3a (90)	tzload (240)

Table 4: Comparing Architectures

3.4 Different Architectures

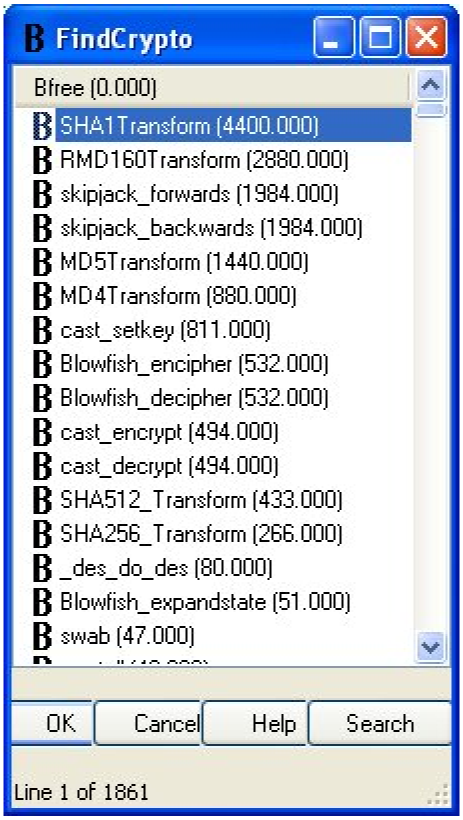
All of the analysis so far has been with the 32bit Intel architecture. The same ideas apply when moving to a new architecture, but some familiarity is required with the instruction set of the target architecture. In this case, *findcrypto* was ported to run on SPARC (Scalable Processor Architecture) version 9 (aka *SPARC64*), and Table 4 details the top 15 findings from running *findcrypto* on the C library from *i386* versus *SPARC64*. The compiler used on both architectures is *gcc* version 3.3.5 on OpenBSD 4.3.

There is some movement of functions in the table between the two architectures, but generally all of the functions are listed. SHA256 and SHA512 do not appear until 17th and 16th, respectively (not shown).

The port to *SPARC64* is relatively new, and it is expected that further refinement will produce more consistent results on this architecture. Lessons learned from this port will be applicable to other processors as well. It is further believed that *findcrypto* can be extended to work with various virtual machine environments like .NET and Java.

3.5 Positive and Negative Examples

The initial implementation of *findcrypto* was targeted at processing *objdump* disassembly of binaries. It has also



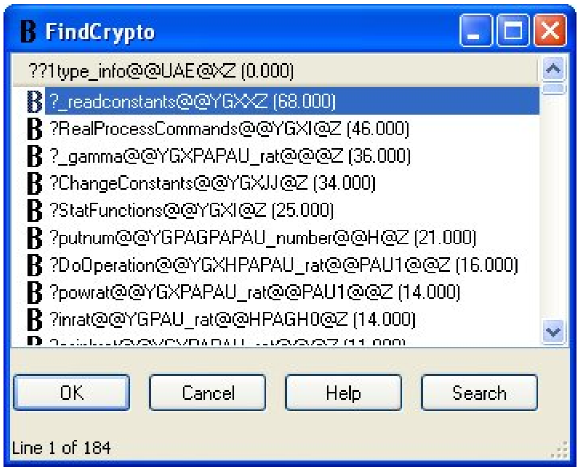
```
Bfree (0.000)
SHA1Transform (4400.000)
RMD160Transform (2880.000)
skipjack_forwards (1984.000)
skipjack_backwards (1984.000)
MD5Transform (1440.000)
MD4Transform (880.000)
cast_setkey (811.000)
Blowfish_encipher (532.000)
Blowfish_decipher (532.000)
cast_encrypt (494.000)
cast_decrypt (494.000)
SHA512_Transform (433.000)
SHA256_Transform (266.000)
_des_do_des (80.000)
Blowfish_expandstate (51.000)
swab (47.000)

Line 1 of 1861
```

Figure 1: Positive Example

been ported to work with the IDA Pro disassembler. It produces a table that can be clicked on to jump to the location of particular entries.

Figure 1 shows the disassembly of a binary linked against a version of the OpenBSD C library. Symbols are left in the binary to demonstrate that *findcrypto* gives a high score to the cryptographic functions listed.

The second example, Figure 2, shows the disassembly of `CALC.EXE`, a popular target for demonstration. This binary should not contain cryptography, and *findcrypto* gives a score of only 68 to the top scoring function. Symbols are left in mangled form.



```
??1type_info@@UAE@XZ (0.000)
?_readconstants@@YGXXZ (68.000)
?RealProcessCommands@@YGXI@Z (46.000)
?_gamma@@YGXPAPAU_rat@@@Z (36.000)
?ChangeConstants@@YGXJJ@Z (34.000)
?StatFunctions@@YGXI@Z (25.000)
?putnum@@YGPAGPAPAU_number@@H@Z (21.000)
?DoOperation@@YGXHPAPAU_rat@@PAU1@@Z (16.000)
?powrat@@YGXPAPAU_rat@@PAU1@@Z (14.000)
?inrat@@YGPAU_rat@@HPAGH0@Z (14.000)

Line 1 of 184
```

Figure 2: Negative Example

4 Conclusions

The *findcrypto* software is effective at locating various block ciphers (DES, CAST-128, SKIPJACK, BLOWFISH, etc.) and hash functions (RMD160, SHA1, SHA2, MD5, MD4, etc.). The block ciphers listed above are all classified as product ciphers, which combine rounds of simple operations like substitution (S-boxes), permutation (P-boxes), and modular arithmetic. Permutation functions commonly consist of operations like bit-wise rotation (ROL and ROR discussed in Sections 2.2 and 2.3), and XOR (Section 2.1) is often used in a modular arithmetic block. Hash functions are built of similar blocks.

Stream ciphers like, RC4, are more difficult to locate using this method. RC4 in particular scores very low with *findcrypto* (100) compared with a score of over 4000 for SHA1. This is due to the way the permutations and the loop around the XOR operation are implemented. RC4 does not consist of a large number of rounds like the product ciphers, instead its algorithm revolves around a pseudo random number generator.

As mentioned earlier, [Gui06a] examines a binary looking at the operands (data section or immediates within instructions) to find the constants used in various algorithms. By changing these constants or obscuring them, this method can be defeated. For example, changing the initial state constants for MD5 or AES would make these algorithms more difficult to locate identify.

findcrypto is not immune to obfuscation techniques either. Many methods for packing and obscuring binaries are available in the malware community. Therefore, *findcrypto* must be used in conjunction with unpacking tools like [Dat05]. The technique in this work is can also fall victim to other obfuscation methods like inserting "dead" code, which produces results that are never used and do not affect the function output. Also the insertion of entire dead functions can add invalid entries to the score. Static and dynamic analysis techniques, like those implemented by [LR08], can help ease the impact of both techniques.

Notably missing from the discussion thus far are asymmetric algorithms like RSA and Diffie-Hellman (DH) key agreement. In both cases, the algorithms rely on relatively simple mathematical operations on large numbers (512 bits and greater). Because the size of the operands will not fit into a normal integer type on modern CPUs, a different data structure is used: *bignum*. The required operations (modular multiplication and exponentiation) are implemented as functions themselves, so the RSA and DH implementations are simply macro operations on *bignums*. It may be possible to detect the macro operations, though. In the current implementation, *findcrypto* gives *bn_mul_part_recursive()* a score of 21. This function forms the basis for modular multiplication, which is in turn the basis for modular exponentiation.

5 Future Work

The current work has focused on the location of cryptographic routines within object code. It is believed that combining this work with a heuristic algorithm identification method would be invaluable.

Also, the current method relies on relatively simplistic opcode matching. With further analysis, blocks of instructions forming a functional block could be matched and weighted. This functional block method may help the location algorithm with different compilers and architectures.

References

[Bil07] Daniel Bilar. Opcodes as predictor for malware. *Int. J. Electron. Secur. Digit. Forensic*, 1(2):156–168, 2007.

[Dat05] Datarescue. *Using the Universal PE Unpacker Plug-in included in IDA Pro 4.9 to unpack compressed executables*, 2005.

[Gui06a] Ilfak Guilfanov. FindCrypt, January 2006. `http://hexblog.com/2006/01/findcrypt.html`.

[Gui06b] Ilfak Guilfanov. FindCrypt2, February 2006. `http://hexblog.com/2006/02/findcrypt2.html`.

[LR08] Eric Laspe and Jason Raber. Deobfuscator: An automated approach to the identification and removal of code obfuscation. In *REcon*, June 2008.

[NESP08] Karsten Nohl, David Evans, Starbug, and Henryk Plötz. Reverse-engineering a cryptographic RFID tag. In 17^{th} *USENIX Security Symposium*, July 2008.