LO! LLVM Obfuscator

An LLVM obfuscator for binary patch generation

Master of Science Thesis

FRANCISCO BLAS IZQUIERDO RIERA

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2014

LO! **LLVM** Obfuscator
An **LLVM** obfuscator for binary patch generation

FRANCISCO BLAS IZQUIERDO RIERA

Examiner: ANDREI SABELFELD
Supervisor: JONAS MAGAZINIUS

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover:
The dragon representing the **LLVM** project having a rusty part of him replaced by a new blurry part.
The picture represents the idea of replacing broken parts of projects with new obfuscated parts.

Department of Computer Science and Engineering
Göteborg, Sweden January 2014

Abstract

As part of this Master's Thesis some patches to **LLVM** have been written allowing the application of obfuscation techniques to the **LLVM** IR. These patches allow both obfuscation and polymorphism which results in code that is both hard to read and different from previous versions. This, makes finding the real changes made between versions harder for the attacker.

The techniques are applied using a function attribute as the seed for the CPRNGs used by the transformations as a source of entropy. As a result it is possible to mark the functions that should be obfuscated in the prototypes allowing the developer to create binaries with the desired amount of changes and a sufficiently large amount of functions that are hard to read and (if the seed is changed) different from previous versions.

In this Master's Thesis the possible ways in which the applied techniques can be "reversed" have been evaluated to be able to compare the resulting code. For this to succeed a transformation able to obtain **LLVM** IR from the resulting binary code is necessary, this was not done as part of this work.

Acknowledgements

Life is all about choices: you exist because at some point of time your parents made a choice on having a child[1]. You are probably reading this text because you have made a choice on that it will be interesting and you likely are who you are because others have influenced your life through their own choices along with yours.

Choices can be good or bad with a whole gamut of grays in the middle. But independently of the result, the path that a choice makes you take is more important and enriching than the choice itself.

Despite I'm the one presenting this work as my Master's Thesis and thus closing a chapter of my life, it wouldn't have been possible for me to do so if some people hadn't chosen to support me in one or another way. This pages will never be enough to thank all of them.

Even worse, though, was making the big mistake of not writing them as I did on my Computer Engineer final project [11], to make up for this, this section will also cover the acknowledgments that weren't done in that publication.

First of all and typical as it may seem I'd like to thank my parents. If they hadn't chosen to have me this thesis nor the project would have existed. Transitively the thanks should expand to their parents and those's parents (and so until the first freewilled being I suppose) for taking similar choices.

Next of course comes the rest of my family, they have chosen to support me in my studies all along and without their help this wouldn't have been possible.

Going back before I even started my project some people I should thank would be Jon Ander Gómez who had arranged the ICPC local programming contests in the UPV as they helped me develop the skills I needed later and Miguel Sánchez and Alberto Conejero who have been amongst the best teachers I had. Also I should thank the ELP group at the DSIC department for giving me a chance of tasting what research felt like back then.

Focusing on my Computer Engineer project I should be thanking Pedro López who pushed for me, Julio Sahuquillo, my examiner at the UPV, Per Stenström, my examiner at Chalmers and Rubén González, my advisor and the biggest influence in the project.

Finally focusing on this actual work I'd have to thank the PaX Team and Anthony (blueness) G. Basile for their great input on the project, Jonas Magazinius for agreeing to be my advisor (and putting up with me all this time), Andrei Sabelfeld for being such a nice examiner and Grim Schjetne for being such a nice opponent despite so little notice. Also thanks to all who attended the presentation and provided input which has helped improve this document.

[1] Or just on not using contraceptives that fateful night and then following with the pregnancy.

But specially, thanks to all of you who hasn't been mentioned on this page, your small contributions are what really made this possible.

Whilst doing this work many things have changed in my life, I have seen the Hackerspace at Göteborg where I'm writing these lines take off, I have started a relationship with a girl, and have met some new friends. I don't know what the future will bring with it as it's really hard to see it now, but I'm quite convinced on what the past has brought thanks to all that people, as this future yet to come wouldn't have been possible without their incredible help.

Thus, to all those who have helped in one way or another to make this possible I can only say ¡Muchísimas gracias desde lo más profundo de mi corazón![2]

Francisco, Göteborg 14/3/2013

[2]Thanks a lot from the bottom of my heart!

Contents

1 Introduction **1**

1.1 Background . . . 1

1.2 Problem statement . . . 3

1.2.1 Goals . . . 3

1.2.2 Delimitations . . . 3

1.3 Thesis Structure . . . 4

2 Definitions **5**

2.1 Compiler . . . 5

2.1.1 Intermediate representation . . . 5

2.1.1.1 Basic block . . . 6

2.1.1.2 SSA . . . 6

2.1.1.3 PHI node . . . 6

2.1.2 Frontend . . . 6

2.1.3 Middle end . . . 6

2.1.4 Backend . . . 7

2.2 **LLVM** . . . 8

2.2.1 Clang . . . 8

2.2.2 DragonEgg . . . 8

2.2.3 **LLVM** IR . . . 8

2.2.4 Evaluation pass . . . 8

2.2.5 Transformation pass . . . 9

2.2.5.1 Optimization transformation . . . 9

2.2.6 DAG . . . 9

2.2.6.1 Legalization . . . 9

2.2.6.2 Register allocation . . . 9

2.3 Code obfuscation . . . 10

2.3.1 Obfuscation transformation . . . 10

2.3.1.1 Control flattening . . . 10

2.3.1.2 Constant obfuscation 10
2.3.1.3 Opaque predicate 11
2.3.2 Polymorphism transformation 11
2.3.2.1 Register swap 11
2.3.2.2 Dead code insertion 11
2.3.2.3 Code reordering 11
2.4 PRNG 12
2.4.1 CPRNG 12
2.5 Encryption algorithm 13
2.5.1 Symmetric key encryption algorithm 13
2.5.2 Block encryption algorithm 13
2.5.3 AES 14
2.5.3.1 CTR mode of operation 14
2.5.3.2 CMAC mode of operation 15

3 Methodology **16**
3.1 Information gathering 16
3.1.1 Related work 16
3.2 Development 18

4 Algorithm Implementation **19**
4.1 Control Flattening 19
4.2 Constant obfuscation 22
4.3 Register Swap 24
4.4 Code reordering 25
4.4.1 Instruction reordering 25
4.4.2 Basic block reordering 25
4.5 CPRNG 28

5 Code design considerations **30**
5.1 Coding conventions 30
5.1.1 Transformation specific conventions 30
5.1.2 Auxiliar library conventions 30
5.2 Design choices 32
5.2.1 AES implementation used 32
5.2.2 **LLVM** transformations 32
5.2.2.1 Transformation parameters 32
5.2.2.2 Transformation implementation 32

6 Transformation implementations **33**
6.1 Obfuscation key 33
6.1.1 addmodulekey 33
6.1.2 propagatemodulekey 33
6.2 Obfuscation 35

6.2.1 flattencontrol . . . 35
6.2.2 obfuscateconstants . . . 35
6.3 Polymorphic . . . 45
6.3.1 bbsplit . . . 45
6.3.2 randbb . . . 45
6.3.3 randins . . . 45
6.3.4 randfun . . . 45
6.3.5 randglb . . . 46
6.3.6 swapops . . . 46

7 Evaluation 52
7.1 Proof: the CPRNG is a good PRNG . . . 52
7.1.1 Determinism . . . 52
7.1.2 Uniformity . . . 52
7.1.3 Independence . . . 53
7.1.4 Function period . . . 53
7.2 Proof: the CPRNG is secure . . . 54
7.2.1 Next bit-test resistance . . . 54
7.2.2 Impossibility of knowing the result if only the state is known . . . 54
7.2.3 State compromise extension resistance . . . 54
7.3 Proof: the key derivation is secure . . . 55
7.4 Reversing the transformations . . . 56
7.4.1 Defining a global ordering of values . . . 56
7.4.2 Defining a global ordering of instructions . . . 56
7.4.3 Reversing the randfun and randglb transformations . . . 56
7.4.4 Reversing the swapops transformation . . . 57
7.4.5 Reversing the randins transformation . . . 57
7.4.6 Reversing the obfuscateconstants transformation . . . 57
7.4.7 Reversing the flattencontrol transformation . . . 57
7.4.8 Reversing the bbsplit transformation . . . 57
7.4.9 Reversing the randbb transformation . . . 57
7.5 Binary patch obfuscation technique evaluation . . . 59

8 Conclusions 60

9 Future development 61

Bibliography 62

A Using the tools 66

B Popularization 69
B.1 Programming computers . . . 69
B.1.1 Assembly . . . 70

B.1.2 Programming languages 70
B.2 Compilers 72
B.3 Antireverse engineering 73
B.3.1 Obfuscation techniques 73
B.3.1.1 Control flattening 74
B.3.1.2 Constant obfuscation 75
B.3.1.3 Register swap 76
B.3.1.4 Instruction Reordering 77
B.3.2 Focused obfuscation 79
B.3.3 Compiler-level obfuscation 79
B.4 Deobfuscation 81
B.4.1 Control unflattening 81
B.4.2 Constant deobfuscation 81
B.4.3 Register swap 81
B.4.4 Instruction Reordering 81
B.5 Program updates 82
B.6 Practical example 83

C Source code 84
C.1 Patches to LLVM 84
C.2 Patches to Clang 85
C.3 Obf library 87
C.4 AES library 116

1

Introduction

1.1 Background

As stated by [5], there is no silver bullet to prevent programmers from making mistakes when coding applications. Some of these errors may not necessarily cause more than a nuisance when hit by the users of the software, but some of them may actually end up being vulnerabilities exploitable by a third party, which can have undesirable effects for the software user ranging from unavailability of software to more serious compromises like attackers gaining control of the system.

Generally, the likelihood of a software error increases with the size and complexity of the software. Likewise, the probability of one of the errors being exploitable increases with the amount of software errors. To make matters worse, the majority of current devices use a reduced set of software programs, due to the size, complexity, and incompatibility of some softwares. For example, according to StatCounter more than half of the opertaing systems (or OS) used to browse the web are Microsoft Windows NT derivatives [24] (mostly Windows 7 and XP), while two-thirds of web browsing is done using either Microsoft Internet Explorer, Google Chrome, or Mozilla Firefox [25]. As a result of this lack of diversity, vulnerabilities can affect large amounts of devices and, since most are connected to the Internet, attackers can remotely exploit these.

Usually, when a vulnerability is discovered or reported to the creator of the affected software, he addresses the issue and creates a new version of that software in which the problem is corrected.

Since distributing an entire copy of the new version of the program may require many resources (For example, the Firefox 27.0.1 xul.dll file containing most of the GUI runtime is 21.7 MB.), in most cases the developer instead releases a small file containing the required updates and, occasionally, a short program to apply these changes to the software. The files with the required changes are generally known as patches, and the process of applying them is known as patching, although this word may also be used to refer to the entire process of correcting a software issue. In addition to addressing vulnerabilities, updating software in this manner is used to correct other types of software errors (known as bugs) and to add new features that improve the software, though the latter is quite rare.

When a developer deploys an upgrade, it necessarily takes some time before all of the users actually apply the patch and secure their systems from attack, even if the update process is done automatically by the software without user intervention. This period, from the time the patch is made public to the time the last user applies it, represents a window in which third parties can attack any non-upgraded software. Furthermore, such parties could infer what vulnerability is present in the (un-patched) system, based on the published patches.

1.2 Problem statement

As patches tend to be small to reduce the amount of resources required by the updating process, it is relatively easy for the attacker to identify what is being changed on the system. Thus, the attacker can discover the fault being fixed and abuse it, if the user has not yet patched their system. This is usually known as a 1-day vulnerability [22].

1.2.1 Goals

The goal of this thesis is to provide a set of tools which can help software developers increase the amount of time an attacker requires to find and understand a particular update in the patch. Two different methods will be combined to achieve this:

1. Applying obfuscation transformations to the code to render it more difficult for a third party to understand.

2. Applying polymorphism transformations to increase the amount of differences between the old and the new program and, thus, decrease the signal to noise ratio.

The project aims are to:

1. Provide a set of tools to allow the application of those transformations without modifying the original code during file compilation.

2. Adapt the polymorphism transformations and obfuscation transformations so they can be applied to a compiler's intermediate representation (or IR), particularly to **LLVM** (whose IR is named LLVM IR) which uses a three way static single assignment form (or SSA).

3. Create a proof of concept for integrating these tools into a real compiler, which would then allow the developer to focus transformations only on the desired functions.

4. Evaluate the effectiveness of the transformations and, if possible, explain the steps that could be taken to reverse them (or, at least, render them useless).

1.2.2 Delimitations

Every project inherently contains certain restrictions due to limited resources. In this project, given the amount of time and other resources available for completion, the following limits were placed upon the goals:

1. Not all of the possible transformations will be adapted or evaluated.

2. A proof of concept for the method in which an attacker could make the transformations useless (if it is possible to do) will not be developed.

1.3 Thesis Structure

As seen above, this work starts with an introduction (chapter 1), containing the project background (section 1.1), problem statement (section 1.2), project goals (subsection 1.2.1), delimitations on those goals (subsection 1.2.2), and thesis structure (section 1.3).

It continues with definitions (chapter 2), in which the various concepts and terms used in this work are presented. This section also serves as an introduction for the less experienced researcher.

From there, this paper proceeds onto explaining the different methodologies utilized during the project (chapter 3). Including the process used to obtain information (section 3.1), an overview of relevant research papers and other current work (subsection 3.1.1) , and, finally, the development techniques and conventions necessary for the project (section 3.2).

In the next chapter (chapter 4) algorithms implemented in this project, along with the cryptographic pseudorandom number generator (or CPRNG), and their conversion into code, are explained.

Afterwards, the various decisions taken while designing the code are outlined (chapter 5). In particular, style decisions (section 5.1), and design decisions (section 5.2).

Next, implementations of the different transformations are discussed in specifics (chapter 6).

Following that, some proofs are provided and the developed techniques are studied (chapter 7). A proof that the developed CPRNG has the properties desired for a pseudorandom number generator (or PRNG) is provided (section 7.1) , along with proof that CPRNG has the properties to be secure (section 7.2). Furthermore, proof that the key generation system used is also safe is shown (section 7.3). Finally, an analysis of how an attacker could reverse the developed transformations is given (section 7.4), and an evaluation of the technique used for obfuscation of binary patches is shown (section 7.5).

This paper ends with the project conclusions (chapter 8) and the expectations of future development based on this work (chapter 9). A list of references can be found afterwards.

In the annexes, instructions on how to use the developed tool (appendix A), an explanation of the project aimed at the general public (appendix B), and project sources and patches (appendix C) are found.

2

Definitions

2.1 Compiler

A compiler is a tool able to perform translations from one language to another, generally from a higher level language to a lower level one that is closer to the language of the destination platform.

When creating a compiler two options can be chosen: making a direct translation between each source and destination, or using a frontend to convert the language to a common intermediate language and then a backend to convert from this common language to the destination language.

The first alternative allows the programmer to exploit more of the expressibility of the destination language and results in more efficient and smaller translations; however, it requires a different compiler for each source and destination pair. The result is that the number of different compilers required to cover a particular set of source and destination languages is the product of the number of those languages.

The second alternative allows the programmer to maintain a common pipeline to optimize the resulting intermediate language and may result in larger and slightly less efficient translations; however, this alternative only requires as many frontends as source languages and as many backends as destination languages.

2.1.1 Intermediate representation

This name is used to refer to a language or set of languages which the compiler will translate the original code to before translating it into the desired final language. It is also used to refer to any code written in any such language.

Generally, the generic optimization transformations are performed in the intermediate representation, as they can then thus be applied to all of the destination languages.

IR is the abbreviation of intermediate representation and is used by **LLVM** to refer to the language utilized in the optimization stages.

2.1.1.1 Basic block

A basic block, or BB, is a set of instructions with a single entry point and a single exit point. As a result, jumps cannot point to the middle of a basic block, and basic blocks cannot contain more than one jump instruction (which is always placed at the end).

2.1.1.2 SSA

SSA is the acronym for static single assignment form. In general, it is used to refer to a property of intermediate representations: that each variable is assigned only once.

2.1.1.3 PHI node

PHI Nodes are used by SSA languages to assign a variable with the appropriate value, based on the basic block from which the node is reached. This allows SSA languages to have some variables' values assigned from other basic blocks when more than one basic block jumps to a particular basic block. This is the case for loops and conditional structures.

2.1.2 Frontend

The frontend is the part of the compiler transforming the original source language into the first intermediate representation used by the compiler pipeline.

The responsibilities of this part of the compiler include checking the validity of the source code, detecting and warning the user of any uncovered errors, performing any source language specific optimizations, and extracting the metadata from the source code for use at later stages.

2.1.3 Middle end

This is the part of the compiler that handles the transformations between intermediate representations either by transforming it into a different version using the same intermediate representation or by generating code using a different intermediate representations.

Optimizing the code returned by the frontend is the main responsibility of this part of the compiler.

2.1.4 Backend

This is the final part of the compiler pipeline and transforms the last intermediate representation into the desired destination language.

The main responsibilities of this part of the compiler are legalizing the code by applying transformations so that it only uses the instructions supported by the target architecture, performing target specific optimizations, allocating the source architecture registers to the different instructions, and emitting them.

2.2 LLVM

THE **LLVM** project aims to provide a compiler framework with various native frontends (C, C++, objective C, opencl, and Haskell, among others), a gcc intermediate representation code frontend (which allows support for Ada, D, and Fortran), and backends for many CPU- and GPU-based platforms (including ARM, x86, x86-64, MIPS, Nvidia's PTX, and ATI's R600).

LLVM uses the frontend to transform the source code into an SSA intermediate representation known as IR, runs the desired optimization and transformation passes over this code, and finally converts the final SSA to a DAG that is passed to the backend for legalization, register allocation, and instruction emission.

A good explanation of the inner workings of the **LLVM** pipeline can be found at [3].

2.2.1 Clang

Clang is a frontend for **LLVM** supporting C-type languages (C, C++, Objective C, OpenCL, etc.). It is the frontend used by default when compiling those languages by the **LLVM** compiler.

2.2.2 DragonEgg

DragonEgg is a frontend that allows the use of most of the gcc frontends with **LLVM** and support of languages like ADA or Fortran.

2.2.3 LLVM IR

LLVM IR is the SSA language used internally by **LLVM** for intermediate representations. This language can be represented by a set of memory structures during compilation time, bitcode when stored on a disk or passed around pipes, and a human readable assembly-like representation useful for developers.

Well formed code using this language must hold at least the properties stated by the Verifier pass. A good description of the language can be found at [17], and the properties held by well-formed language instances are explained in the Verifier.cpp file.

2.2.4 Evaluation pass

An evaluation pass parses the IR to generate some information about the code that can be used by other passes.

Evaluation passes cannot modify the IR.

2.2.5 Transformation pass

A transformation pass parses the IR and generates a new IR (and, occasionally, information about the generated IR). In general, these passes must generate a functionally equivalent IR in order to be considered correct. This is the type of optimization passes used by **LLVM**, obfuscation passes, and polymorphism passes that have been created during this project.

2.2.5.1 Optimization transformation

An optimization transformation is a transformation pass which parses the IR and generates a new IR (and, occasionally, some information about the generated IR). These transformations generate a functionally equivalent IR which is expected to execute using less resources from the system.

2.2.6 DAG

DAG is the acronym for directed acyclic graph. DAGs are the intermediate representation passed to the backends for transformation into the target instructions.

2.2.6.1 Legalization

Legalization is the process by which the backend transforms unsupported instructions in the DAG into instructions supported by the instruction emitter. For example, this stage would convert floating point instructions in an architecture into calls to auxiliary functions that support them or sets of integer instructions able to perform the same operations.

2.2.6.2 Register allocation

This is the process by which the registers available in the target architecture are assigned to be source and destination registers for the DAG instructions. Performing this process properly will result in significant performance differences in the resulting code, especially for architectures with a small number of general purpose registers.

Finding the optimal allocation for registers can be reduced to graph coloring, which is known to be an NP-Complete problem; however, a proof exists in [10] demonstrating that register allocation can be done in polynomial time when using SSA form.

2.3 Code obfuscation

The procedure by which the input source code is transformed into a harder-to-read code which is functionally equivalent to the source code is called code obfuscation. Unlike optimization transformations, obfuscation transformations do not necessarily produce faster code, as they only focus on making the resulting code harder for humans to understand. For example the control flattening code reduces the efficiency of the code, as it creates more difficulties for the jump predictor. In a similar way, different constant obfuscation techniques make the processor execute a greater number of instructions to achieve the same results.

2.3.1 Obfuscation transformation

Obfuscation transformations are transformations which perform code obfuscation. This term will be used to refer only to those code obfuscation transformations which are not intended to generate polymorphic code, such as constant obfuscation and control flattening although in other literature the term “obfuscation transformations” covers a wider variety of transformations intended to make the resulting code harder to read or to detect.

2.3.1.1 Control flattening

The control flattening transformation was initially defined by Wang [32]. It is generally based on the idea of picking two or more basic blocks and making them jump unconditionally to a new basic block, where the destination basic block will be chosen depending upon the previous basic block. In this project, the transformation is applied to all of the basic blocks inside a function, resulting in a single main basic block which decides where to jump at its completion. The details of the algorithm implementing this transformation with **LLVM** IR will be explained in the implementation section.

2.3.1.2 Constant obfuscation

The constant obfuscation transformation intends to make constants harder to read by transforming them into a set of instructions which will produce the desired constant. There are multiple ways of achieving this:

1. Fetching the constants from a memory position, for example an array. [32] proposed using aliasing transformations over the array to make reversing the transformation more difficult (although this last part has not been implemented in this particular project).

2. Encrypting the constants, for example with arithmetic operations, and converting them into a cipher text and a key which when combined will result in the desired constant.

3. Using an opaque predicate which will result in the desired constant. This was not implemented in this project, either.

2.3.1.3 Opaque predicate

An opaque predicate is a predicate which will return the same value independently of the input values. These are usually derived from mathematical identities.

2.3.2 Polymorphism transformation

This kind of transformation chooses and performs one of many possible transformations of a type which can be applied to the source code. By changing the seed of the CPRNG used by these transformations different instances of code functionally equivalent to the source code are created, which can be helpful in making the set of differences of a patch larger.

2.3.2.1 Register swap

This transformation works by changing one general purpose register to another in the code. This results in different instructions depending on the architectural register being used.

2.3.2.2 Dead code insertion

Dead code insertion consists of inserting code that is unused by the resulting program. This dead code may even be executed by the program, but the code's results are not used by the program.

2.3.2.3 Code reordering

Code reordering changes the order of the code inside a program, resulting in multiple different programs depending upon how the code is reordered.

2.4 PRNG

A PRNG or PseudoRandom Number Generator is a piece of code used to generate a series of numbers which has properties similar to those of real random numbers. Since PRNGs generate the pseudorandom numbers by maintaining a state derived from an initial seed (which is a small number used to initialize the PRNG), it should be noted that they are deterministic, as they will generate the same sequence when given the same seed and thus produce reproducible results.

Good PRNGSs follow these properties [8]:

Determinism Given the same seed the PRNG will produce the same sequence of numbers.

Uniformity In the sequence all numbers are equally probable.

Independence Each output does not appear to depend on previous ones.

The previous properties, in turn, cause the following properties:

Good distribution The outputs are evenly distributed along the sequence.

Good dimensional distribution The outputs are evenly distributed when analyzed over many dimensions.

Appropriate distance between values The distance between the values generated is similar to that of a real random number generator.

Long period The generator requires a large amount of iterations before ending up in the same state (and thus generating the same sequence again).

An example of PRNG is the rand() function provided by the C library (seeded by the srand() function).

2.4.1 CPRNG

A CPRNG for Cryptographic PseudoRandom Number Generator, or CSPRNG for Cryptographically Secure PseudoRandom Number Generator, is a PRNG with some added properties that make it more resistant to cryptanalysis.

CPRNGs satisfy the properties of a good PRNG, while also holding the following, stronger properties:

Satisfy the next-bit test Given the first k bits of the sequence, there is no polynomial time algorithm able to predict the next bit with more than 50% accuracy.

Withstand state compromise extensions If the state of the CPRNG, or part of it, has been compromised it should be impossible to guess the previous values returned by the generator.

2.5 Encryption algorithm

An encryption algorithm is an algorithm, which given some data and a key, merges the data with that key such that someone without the correct key cannot read the data being encrypted with the algorithm.

Although there are some non-standard hieroglyphs carved in Egypt around 1900 BC that were initially suspected of being the earliest cryptography, these are now considered to be written merely as entertainment for literate individuals. Thus, the first verified cryptography use dates to 1500 BC when an encrypted Mesopotamian clay tablet containing some recipes, considered trade secrets, was written.

Despite this early start, not much serious work on cryptography and cryptanalysis was done until the last century, when computers were used to automate the processes.

Perhaps the most robust encryption technique is the encryption algorithm known as a one-time pad, which when used correctly is unbreakable, as the entropy provided by the key (if truly random) is equal to the entropy of the message. Thus, it is impossible to derive any information from the message.

There exist some ancient encryption algorithms, like Caesar or Vigénere ciphers, but DES, AES, RSA, and RC4 are more recent examples.

2.5.1 Symmetric key encryption algorithm

A symmetric key encryption algorithm is an encryption algorithm which uses the same key for encryption and decryption of data, so that key must be protected from outsiders in order to protect the data.

Asymmetric key encryption algorithms are the opposite of symmetric key encryption algorithms. With asymmetric key encryption algorithms, data is encrypted with one key and decrypted with another, and there is no easily computable way of getting from the encryption key to the decryption one. As a result, the encryption key can be published and is called a public key, whilst the decryption key is kept secret by the receiver of the message and is called a private key.

Classical ciphers, and modern ones like DES, AES, and RC4, are symmetric key encryption algorithms.

2.5.2 Block encryption algorithm

A block encryption algorithm is an encryption algorithm which works over fixed-size groups of bits independently.

Generally, these can be considered a pseudo random permutation, which is a function performing a 1-to-1 mapping of an n bit input into an n bit output where the key is used to choose one of the possible mappings.

As an encryption algorithm only able to encrypt a particular block size is quite unusable per se, various modes have been developed for these types of encryption algorithms to make their use easier. The simplest such example is ECB, in which the message is divided into blocks and then encrypted with the same key. Sadly, this is also quite insecure, as equal blocks will be encrypted into the same output.

To solve this security problem, advanced modes like CBC (where the previous result is mixed with the plaintext before encryption) are available. Even more advanced modes, such as those used to convert block encryption algorithms into stream ciphers (which encrypt single bits) or authentication algorithms, or to provide authenticated encryption (with or without authenticated extra data), also exist. The security of most of these modes is usually based on the assumption that the algorithm is a pseudorandom permutation.

Examples of such ciphers are AES and DES.

2.5.3 AES

Advanced Encryption Standard (or AES) is the NIST standardized version of Rijndael, the winner of the AES selection process. It is a symmetric key block encryption algorithm with a block size of 128 bits and key sizes of 128, 192, and 256 bits.

The AES competition was held to choose an appropriate successor to DES, the previous NIST standardized symmetric key block encryption algorithm which had key sizes of 56 bits and block sizes of 64 and could be attacked by brute force.

Thanks to the standardization and widespread use of AES, efficient free software implementations exist along with very efficient hardware implementations, including those which use the AESNI instruction set on newer, x86 processors.

2.5.3.1 CTR mode of operation

The CTR (or counter) mode of operation converts a block encryption algorithm into an stream encryption algorithm by using it to encrypt blocks which contain an increasing counter and then doing an xor operation of the results with the plaintext, as would be done by any stream encryption algorithm. This provides some advantages, such as easy parallelization of the algorithm when applying it to various blocks, and, as with any stream encryption algorithm, padding is not needed.

The CTR counter can be implemented in many ways (for example, by multiplying a non-zero block number by a prime number), but the most popular method is to apply an increment of one each time to the unsigned integer number made from the bits in the previously used block, as this method is simple (especially when using a carry-aware addition instruction) and still secure.

2.5.3.2 CMAC mode of operation

The Cipher based Message Authentication Code (or CMAC) mode of operation is an authentication mode for block encryption algorithms which generates an authentication tag for a given input. This mode of operation is similar to how a keyed hash based authentication algorithm would work.

When used with a secret key, CMAC mode will prevent any information in the message from being derived from the authentication tag, as long as the key is not known and the block encryption algorithm is secure. If used with a known key, though, in some cases CMAC mode can be reversed by the method shown in [12], but it is still useful as a simple entropy collection algorithm for key derivation from variably sized data.

3

Methodology

3.1 Information gathering

INFORMATION gathering has been conducted mainly by electronically searching for published papers and other online sources that address the desired concepts, as well as through reviewing the citations of those documents to discover other interesting, related material.

The main focus has been on researching obfuscation transformations and polymorphism transformations that could be applied to this project. In this search, [1] and [33] have been of special interest, given the outlooks they provide.

Some of the keywords used when searching for information have been *Code Obfuscation*, *Control Flattening*, *Opaque Predicate*, *Binary Obfuscation*, *Dissasembly*, *1-day Exploit*, *Metamorphic Code*, *Deobfuscation*, and *Code Transformation*.

3.1.1 Related work

Quite a lot of research has already been done on the topic of code obfuscation, even though [2] proved that some functions cannot be obfuscated. One of the most relevant papers on the topic is [32], which in Chapter 4 introduces a set of obfuscation techniques that was later further developed by [33] into a general obfuscation method. This method has served as the basis for the obfuscation techniques implemented in this project.

On a similar topic, [15] proposes the insertion of junk bytes before basic blocks along with the use of opaque predicates in the added branch instructions to make it more difficult for dissasemblers to go back to the original code by disrupting the instruction stream.

The effectiveness of these techniques has been analyzed in papers such as [21], which also contains an overview of some of these techniques and concludes that they can be applied in an effective form at source code level. There has also been research on reversing these techniques at [28, 20, 19], which introduce certain automatic and semi-automatic tools to help in deobfuscating code generated through some common methods, including control flattening.

Since obfuscation allows the intentions of the code being executed to be hidden, it should not come as a surprise that one of the main focuses of obfuscation has been its use in Malware programs in order to make such programs harder to detect, as shown by papers such as [4, 26]. Even though the techniques explained in those works are useful for similar purposes to that of this project, the focus of this project is very different.

Moving into more recent research, a robust review of available obfuscation techniques can be found at [34]. One of the most recent and promising developments in this field is the technique pointed to in [7], which proposes the use of cryptography to hide code functionallity. Finally, some development on obfuscation using LLVM is presented by [14, 6, 26, 23].

There exist many solutions to allow code obfuscation, but none could be found which allow for focusing the transformations on the desired functions. Such an obfuscator would permit the developer to control the size of the final patch. Additionally, many tools only focus on obfuscating the resulting binary code, but in order to improve code portability the developed tool needs to work on an upper layer. In this project the focus will be on the intermediate representation code, resulting in a more portable tool.

The most similar project to the one described in this paper is obfuscator-llvm [13] [1]. This project is limited by the need to identify functions by name (such approach is not adequate in some languages, such as C++ where mangling is used); the use of a CPRNG seeded randomly (which will result in different code on every compilation, making patch generation more difficult); and a control flattening implementation that heavily depends upon memory accesses (which require later optimization).

[1] Code is available at `https://github.com/obfuscator-llvm/obfuscator`

3.2 Development

For this project the **LLVM** compiler and the Clang frontend were chosen because of the quality of their documentation (especially due to their explicative tutorials, available at [18] and [27]).

As a result, the language used when developing this system was C++, as it is the same language used by the above two projects. Notably, though, the AES library, which provided the required AES modes for the CPRNG derived from Gladman's library [9], is written in C.

Since one of the objectives of this project is to produce code that can one day be merged with **LLVM**, development has been made against the subversion sources, as was the case for Clang[2].

[2]The patches to the sources are all available for reference at http://klondike.es/programas/llvm_obf/

4

Algorithm Implementation

4.1 Control Flattening

CONTROL flattening tries to ensure that all basic blocks end up on the same basic block, where the choice for the next basic block will be made based on the information provided by the previous basic block (including which basic block it was).

Depending upon the terminating instruction type, different actions will be needed on the end of the basic block before jumping to the common basic block. Unconditional branches should only transfer their value to a PHI node on the common basic block, while conditional branches can transfer the value of a select instruction to that PHI node. Advanced constructs, such as a switch, can be implemented in a similar way. Regardless, because instructions such as indirect jumps cannot be easily handled, the basic block may always be split before the terminator instruction so that it is processed as an unconditional branch.

In this project, this transformation is implemented with the algorithm shown at algorithm 4.1.

In order to improve the obfuscation generated by this transform, a pass which randomly splits basic blocks can be used to make the basic blocks smaller and thus harder to follow. Such an algorithm is defined at algorithm 4.3.

Algorithm 4.1 Control Flattening

```
function controlFlattening(F)
    if not entryblock E of F contains only a non conditional jump then
        create a block B with a non conditional jump to E
        make B the entryblock of F
    end if
    create a block M
    for all instruction I_1 in F do
        replaceUses(F, I_1)
    end for
    for all PHI node P in F do
        if not P is on M then
            move P to M
            for all basic block B not defined on P do
                make P be P on B if P is alive
            end for
        end if
    end for
    for all terminator instruction I in F do
        if not I can be processed then
            split the basic block containing I before I
        end if
    end for
    create an empty map I of identifiers to basic blocks
    for all basic block B in F do
        if B is reachable from M then
            add an unique identifier for B on I
        end if
    end for
    create a PHI node P on M
    for all basic block B in F do
        if B will point to M then
            make P be the appropriate value of I on B
            change the terminator instruction so it points to M
        end if
    end for
    create a switch instruction in M
    make M jump depending on the value of P
end function
```

Algorithm 4.2 Use replacement

```
function replaceUses(F, I_1)
    create an empty queue Q
    for all instruction I_2 in users of I_1 do
        if I_2 is a PHI node then
            if I_2 gets the value I_1 from a block not containing I_1 then
                queue I_2 on Q
            end if
        else if I_2 is on a different basic block than I_1 then
            queue I_2 on Q
        else if I_2 is a terminator we can't process then
            queue I_2 on Q
        end if
    end for
    if Q contains elements then
        create a new PHI node P in M
        make P be P on blocks without I_1 where I_1 is alive
        make P be I_1 on the block with I_1
        for all instruction I_2 in Q do
            replace all uses of I_1 in I_2 by P
        end for
    end if
end function
```

Algorithm 4.3 Block Splitting

```
function blockSplitting(F, X, Y)
    for all instruction I in F do
        if x with Pr(x = true) = X/Y then
            split the basic block containing I before I
        end if
    end for
end function
```

4.2 Constant obfuscation

Constant obfuscation is implemented by replacing constants with a set of instructions that result in the original constant. This can only be implemented when the instruction containing the constant to be replaced is capable of using the result of other instructions in place of a constant at that particluar position. The algorithm used for constant obfuscation is defined at algorithm 4.4.

The algorithm to obfuscate constants is implemented at algorithm 4.5. Currently, this algorithm is only utilized for integer constants, as their arithmetic is relatively easy to predict. For additional simplicity, some of the variables from the parent function are not passed on to child functions in the pseudocode.

Algorithm 4.4 Finding constants to obfuscate

```
function constantObfuscation(F, X, Y)
    create a pointer P_A to the array with the constants moved to memory
    for all basic block B in F do
        for all PHI node P in B do
            for all Constant C in P do
                set I_P to the terminator of the block returning C
                replace C with obfuscateConstant(C, I_P)
            end for
        end for
        for all Instruction I in B do
            for all Constant C in I do
                replace C with obfuscateConstant(C, I)
            end for
        end for
    end for
    generate the array A with constants moved to memory
    point P_A to A
end function
```

Algorithm 4.5 Obfuscating a constant

```
function obfuscateConstant(C, I)
    if not C is integer then
        return C
    end if
    if size of C ≤ integer array element size then
        if x with Pr(x = true) = 1/2 then
                                                    ▷ Memory fetch algorithm
            create a constant C_1 with the current size of the constant array
            push C to the end of the constant array
            if x with Pr(x = true) = X/Y then
                replace C_1 with obfuscateConstant(C_1, I)
            end if
            insert before I a load instruction L_1 of the array address from P_A
            insert before I a displacement instruction D with C_1 over L_1
            insert before I a load instruction L_2 of D
            return L_1
        end if
    end if
                                    ▷ Equivalent arithmetic instruction algorithm
    create a random constant C_1
    if x with Pr(x = true) = X/(2Y) then
        replace C_1 with obfuscateConstant(C_1, I)
    end if
    choose randomly an operation O of xor, add or sub
    create a constant C_2 so C_1(O)C_2 = C
    if x with Pr(x = true) = X/(2Y) then
        replace C_2 with obfuscateConstant(C_2, I)
    end if
    insert before I a O instruction O_i with operands C_1 and C_2
    return O_i
end function
```

4.3 Register Swap

Since it is heavily architecture dependent, it would be quite complicated to define this transformation without working directly with the architectural DAG. The reason for this is that the **LLVM** IR has an unlimited number of anonymous registers, thus making it impossible to swap two registers without also swapping the instructions, which could lead to execution order issues.

To avoid this pitfall and gain a small portion of functionality, this project implements random swapping of the operands of binary operators, where possible. The idea behind this is that the register allocator may decide to issue the registers in a different order when processing the DAGs. Furthermore, the effect of this register swapping is later improved by the code reordering transformation, which takes into account instruction dependencies inside basic blocks to ensure properly kept ordering.

The algorithm used for this simple transformation is defined at algorithm 4.6.

Algorithm 4.6 Register Swap

```
function registerSwap(F)
    for all instruction I in F do
        if I has 2 operands and I is conmutative then
            if x with Pr(x = true) = 1/2 then
                swap operands 1 and 2 of I
            end if
        end if
    end for
end function
```

4.4 Code reordering

CODE reordering inside functions is mainly implemented through randomly reordering the basic blocks and the instructions inside the function. The algorithm to do so has some peculiarities, defined in the following sections.

4.4.1 Instruction reordering

Instruction reordering requires instructions with side effects to always be executed in the same order (as the effects can cause hidden dependencies). It also requires dependencies to be executed before the instructions which use them. Reordering is applied on a per basic block basis to reduce the scope of the pass, as jumps would make the process more complicated. The full algorithm presented at algorithm 4.7 is dividided into PHI node scheduling (presented at algorithm 4.8), instruction dependecy list generation (presented at algorithm 4.9), and instruction scheduling (presented at algorithm 4.10).

PHI nodes are handled independently from the other instructions, because they have no dependencies between them and must always be scheduled at the begining of the basic block. Furthermore, the terminator instruction is not altered, as it must always be at the end of the basic block.

4.4.2 Basic block reordering

Basic block reordering only requires that the entry basic block is kept the same. The algorithm simply creates a new ordering of all of the basic blocks on the function (except the entry basic block) and reorders them according to that arrangement. The algorithm definition can be seen at algorithm 4.11.

Algorithm 4.7 Instruction Reordering

function INSTRUCTIONREORDERING(B)
  PHISCHEDULING(B)
  store in L and M the return value of INSTRUCTIONDEPENDENCIES(B)
  INSTRUCTIONSCHEDULING(B, L, M)
end function

Algorithm 4.8 Schedule PHI nodes

```
function phiScheduling(B)
    make P_1 the first PHI node in B
    make L a list containing all PHI nodes in B
    while size(L) > 0 do
        extract a random element P_2 from L
        if not P_2 = P_1 then
            swap the PHI nodes P_1 with P_2
        end if
        make P_1 point to the PHI node after P_1
    end while
end function
```

Algorithm 4.9 Instruction dependency list creation

```
function instructionDependencies(B)
    make L an empty list
    make M a map of instructions to instruction lists
    for all instruction I in B except PHI nodes and terminators do
        if I has side effects then
            for all instruction I_2 in B after I do
                if I_2 has side effects or I_2 reads memory then
                    append I to M[I_2]
                end if
            end for
        else if I reads memory then
            for all instruction I_2 in B after I do
                if I_2 has side effects then
                    append I to M[I_2]
                end if
            end for
        end if
        for all operand O in I do
            if O is an instruction in B before I then
                append O to M[I]
            end if
        end for
        if M[I] is empty then
            append I to L
        end if
    end for
    return L, M
end function
```

Algorithm 4.10 Schedule Instructions

```
function instructionScheduling(B, L, M)
    make I_1 the first intruction not being a PHI node in B
    while size(L) > 0 do
        extract a random element I_2 from L
        if not I_2 = I_1 then
            swap the instruction I_1 with I_2
        end if
        for all dependent instruction D in I_2 do
            Remove I_2 from M[D]
            if M[D] is empty then
                place D on L
            end if
        end for
        make I_1 point to the instruction after I_1
    end while
end function
```

Algorithm 4.11 Basic block reordering

```
function basicBlockReordering(F)
    make B_1 the entry block in F
    make L a list containing all PHI nodes in B
    remove B_1 from L
    while size(L) > 0 do
        make B_1 point to the basic block after B_1
        extract a random element B_2 from L
        if not B_2 = B_1 then
            swap the basic blocks B_1 with B_2
        end if
    end while
end function
```

4.5 CPRNG

Many of the transformations depend upon an entropy source to make random choices. This project uses a CPRNG for this purpouse. The CPRNG utilizes the pad used for encryption by AES in the CTR mode (which is the same as encrypting blocks made of 0s using CTR), skipping any remaining bits until the end of the basic block. This requires a key and an IV. The key is derived by adding data dependent upon the module, the function, and the transformation, in order to prevent the state from repeating. The initial IV is simply a string of 0 bits (as the algorithm will still be safe even if the IV is known). The pseudocode for the CPRNG is provided at algorithm 4.13.

In order to generate the key for this process, we will first summarize the obfuscation key by using CMAC and the key "ABADCEBADABEBEFABADAACABACABECEA". (This is the Spanish phrase "Abad, cebada bebe, fabada acaba, cabecea", which translates to "The abbot drinks barley (referring to beer), ends with the fabada (a Spanish dish made with white beans, sausages, and pork served with the water they were boiled in), thus nods (out of sleepiness).") Afterwards, we will use the resulting key and CMAC to summarize the rest of the metadata which is considered to be public knowledge. This procedure is chosen because it makes it more difficult to retrieve the obfuscation key even if AES is broken and allows usage of any kind of data as an obfuscation key. The algorithm for generating this key is provided at algorithm 4.12.

A proof for the security of a CPRNG created in this way is provided later in this work.

Algorithm 4.12 CPRNG initialization

function CPRNGInitialization(O)
▷ O is the obfuscation key
make K_1 be 0xABADCEBADABEBEFABADAACABACABECEA in big endian
make K_2 the result of $\text{CMACAES}_{K_1}(O)$
if The pass applies to a function **then**
make P a byte set to 1
append to P the module name
append to P a byte set to 0
append to P the function name
append to P a byte set to 0
append to P the pass identifier
append to P a byte set to 0
else if The pass applies to a module **then**
make P a byte set to 2
append to P the module name
append to P a byte set to 0
append to P the pass identifier
append to P a byte set to 0
else
fail as this is not implemented
end if
make K_3 the result of $\text{CMACAES}_{K_2}(P)$
make S be K_3 as key and a string of 0s as IV
return S
end function

Algorithm 4.13 CPRNG usage

function CPRNGRandom(S)
make K the key in S
make I the IV in S
make R the result of $\text{AES}_K(I)$
increase I by 1
store in S the new value of I
return R
end function

5

Code design considerations

5.1 Coding conventions

THE following conventions apply to all of the code which was written for this project, though certain modifications of these were required, given the nature of the original code. Such modifications are explained later.

Code is indented using 4 spaces for each opened brace not yet closed. No new line is inserted between keywords or expressions and opening braces.

Variables and arguments can be named as desired. In general, iterators are either given a letter starting from "i" or defined as "i" followed by an abbreviation of the class being iterated. This convention was chosen mainly to reduce the development time of the PoC, in spite of the maintenance cost, and will probably be dropped if the code is submitted upstream.

5.1.1 Transformation specific conventions

Classes, methods, and functions follow mostly **LLVM**'s conventions: classes use camel case starting with an upper-case letter, and methods and functions use camel case starting with a lower-case letter.

5.1.2 Auxiliar library conventions

Classes, methods, and functions are given names in underscore-separated characters, with case depending upon the use of abbreviations or words. Classes start with an upper-case letter, while methods and functions do not. This will most likely be refactored to adjust to **LLVM**'s conventions in later iterations, although the conventions will be kept on the AES code unless it is merged into the utility library.

5.2 Design choices

A set of libraries and a framework to implement the code needed to be chosen. For AES support, a slightly modified version of Brian Gladman's AES library [9] was selected. For the transformations, **LLVM**'s framework was chosen. In the following subsections the implications of such choices are exposed.

5.2.1 AES implementation used

Brian Gladman's AES implementation was adapted (by altering the CTR mode so that it will only provide the pad) and utilized because of its liberal license and high quality, demonstrated by references to it in Intel's documentation, amongst others [35].

5.2.2 LLVM transformations

LLVM transformations inherit from ModulePass [30] and FunctionPass [29] and are implemented in anonymous namespaces to prevent pollution. (Common code was moved to the Utils.cpp file and implemented in the Obf namespace.)

Transformations are declared by using the RegisterPass [31] template. Also, a per module ID (depending on the class) is declared as it used later for pass identification.

When possible, the transformation keeps the analysis produced and reports it to the pass manager.

5.2.2.1 Transformation parameters

Parameters are passed by the command line and parsed through the cl [16] API in **LLVM**. A specific parser for probabilities was written for this project. Probabilities are defined as "numerator/denominator". For example, a probability of 50% (1 in 2) would be expressed as 1/2.

5.2.2.2 Transformation implementation

The implemented transformations depend upon the presence of an obfuscation key in order to work. As such, the presence of this key is used to decide whether or not the chosen transformations should be applied to a particular function or module.

6

Transformation implementations

6.1 Obfuscation key

THESE transformations handle the obfuscation keys used by other transformations. Some transformations require a module key which can only be provided with the transformations below, whilst others require function keys which can be forced on all functions with these transformations.

6.1.1 addmodulekey

The addmodulekey transformation simply attaches the specified obfuscation key (as named metadata) onto the module for future use by other transformations. A pseudocode definition is provided at algorithm 6.1.

The addmodulekey transformation is the only current means of expressing a module obfuscation key.

The key can be defined using the modulekey parameter, followed by the string used as the module key.

6.1.2 propagatemodulekey

This transformation propagates the module obfuscation key to all of the functions in the current module. It will overwrite any key already in place. A pseudocode definition is provided at algorithm 6.2.

Propagating the module obfuscation key is useful for testing, applying transformations automatically in certain cases, and as an all-or-none switch.

Algorithm 6.1 addmodulekey

```
function ADDMODULEKEY(Module & M, Key K)
    setModuleMetadata(M,"ObfuscationKey",K)          ▷ Set the module key
end function
```

Algorithm 6.2 propagatemodulekey

```
function PROPAGATEMODULEKEY(Module & M)
    String K = getModuleMetadata(M,"ObfuscationKey")
    for all Function F in M do
        setFunctionAttribute(F,"ObfuscationKey",K)    ▷ Set the function key
    end for
end function
```

6.2 Obfuscation

These transformations take the original code and return a new one which is harder for humans to read, yet still functionally equivalent to the original. They are mostly based on the ideas of [32].

6.2.1 flattencontrol

This transformation applies the control flattening algorithm, but it is quite complex given the way in which the **LLVM** IR language is implemented.

Furthermore, the current implementation could benefit from more code modularization. This was not performed, due to the time constraints of the project.

The pseudocode for the transformation is provided at algorithm 6.3.

6.2.2 obfuscateconstants

This transformation applies the constant obfuscation algorithm.

One of the main issues is that some **LLVM** instructions and calls to intrinsics contain operands which must be a constant (for example, the alignment in a load instruction or the destinations on a switch instruction). These constants cannot be replaced by code which returns them.

The current implementation is capable of separating the different transformations into their own modules for simplicity, but this was sacrificed in order to speed up development of the PoC.

The move to an array method could add random data when expanding the constants to make inferring the size more difficult, or the move could use a single byte constant so that bigger constants would be divided into smaller ones and reassembled. Also, randomly reordering the array would make the resulting array impossible to read.

The pseudocode for the transformation is provided at algorithm 6.12.

Algorithm 6.3 flattencontrol

```
function FLATTENCONTROL(Function & F)
    CPRNG R = PRNG(F,"flattencontrol")
    if not ISNULL(R) then
        PREPAREENTRIESANDEXITS(F)
        BasicBlockList L = GENERATENODELIST(F)
        BasicBlock U = GETUNREACHABLE(F)
        BasicBlock M = new BasicBlock                ▷ Create the main node
        APPEND(F,M)
        GENPHINODES(F, M, L)
        MOVEPHINODES(F, M, L)
        REMOVEUNHANDLEDTERMINATORS(L)
        BasicBlock2IntegerMap D = GENERATEBLOCKIDS(L, R)
        HANDLETERMINATORS(L, M, D)
    end if
end function
```

Algorithm 6.4 prepareEntriesAndExits

```
function PREPAREENTRIESANDEXITS(Function & F)
    UNIFYFUNCTIONEXITNODES(F)              ▷ Merge all exit points of the function
    BasicBlock E = GETENTRYBLOCK(F)
    Terminator T = GETTERMINATOR(E)
    if SIZE(E) ≠1 or ISUNCONDITIONALBRANCH(T) then
        InsertionPoint B = BEGIN(E)
        SPLITAT(E,B)              ▷ Make the entry block only an unconditional branch
    end if
end function
```

Algorithm 6.5 generateNodeList

```
function GENERATENODELIST(Function F)
    BasicBlockList L = new BasicBlockList
    for all BasicBlock B in F do
        APPEND(L,B)
    end for
    return L
end function
```

Algorithm 6.6 getUnreachable

```
function getUnreachable(Function & F)
    for all BasicBlock B in F do
        Terminator T = getTerminator(B)
        if isUnreachable(T) then
            return B                              ▷ Return the found block
        end if
    end for
    BasicBlock B = new BasicBlock                 ▷ Create and return a new block
    Unreachable U = new Unreachable
    append(B,U)
    append(F,B)
    return B
end function
```

Algorithm 6.7 genPHINodes

```
function genPHINodes(Function & F, BasicBlock & M, BasicBlockList L1)
    for all BasicBlock B1 in F do
        for all Instruction I in B1 do
            UserList L2 = new UserList                         ▷ Keep cross block uses
            K = false                                          ▷ Shall we keep the value
            for all User U in I do
                if isPHINode(U) then
                    if getBlockForUse(U) ≠ B1 then
                        K = true
                        append(L2, U)
                    end if
                else if isInstruction(U) and getParent(U) ≠ B1 then
                    K = true
                    append(L2, U)
                else if isTerminator(U) and not isBranch(U) then
                    append(L2, U)
                end if
            end for
            if not empty(L2) then
                PHINode P = new PHINode
                append(P,M)
                for all BasicBlock B2 in L1 do
                    if B2 = getEntryBlock(F) then
                        valueFrom(P,B2,undefined)
                    else if B2 = B1 then
                        valueFrom(P,B2,I)
                    else if K then
                        valueFrom(P,B2,P)
                    else
                        valueFrom(P,B2,undefined)
                    end if
                end for
                for all Use U in L2 do
                    replaceUseWith(U,P)
                end for
            end if
        end for
    end for
end function
```

Algorithm 6.8 movePHINodes

```
function movePHINodes(Function & F, BasicBlock & M, BasicBlockList L)
    for all BasicBlock B_1 in F do
        for all PHINode P_1 in B do
            PHINode P_2 = new PHINode
            append(P,M)
            for all BasicBlock B_2 in L do
                if hasValueFrom(P,B_2) then
                    V = getValueFrom(P,B_2)
                    valueFrom(P,B_2,V)
                else if B_2 = getEntryBlock(F) then
                    valueFrom(P,B_2,undefined)
                else
                    valueFrom(P,B_2,P)
                end if
            end for
            for all Use U in P_1 do
                replaceUseWith(U,P_2)
            end for
        end for
    end for
end function
```

Algorithm 6.9 removeUnhandledTerminators

```
function removeUnhandledTerminators(BasicBlockList L)
    for all BasicBlock B in L do
        T = getTerminator(B)
        if not isBranch(T) then
            splitAt(B, T)
        end if
    end for
end function
```

Algorithm 6.10 generateBlockIds

```
function generateBlockIds(BasicBlockList L, CPRNG & R)
    BasicBlockSet S = new BasicBlockSet
    for all BasicBlock B in L do
        T = getTerminator(B)
        for all BasicBlock B in getDestinations(T) do
            add(S, B)
        end for
    end for
    BasicBlockArray A = toArray(S)
    randomizeOrder(R, A)
    Integer P = 0
    BasicBlock2IntegerMap D = new BasicBlock2IntegerMap
    for all BasicBlock B in A do
        D[B] = P
        P = P + 1
    end for
    return D
end function
```

Algorithm 6.11 handleTerminators

```
function handleTerminators(BasicBlockList L, BasicBlock & M, BasicBlock2IntegerMap D)
    PHInode P = new PHInode
    append(M, P)
    for all BasicBlock B in L do
        T = getTerminator(B)
        remove(B, T)
        if isConditionalBranch(T) then
            C = getCondition(T)
            D_t = getDestinationTrue(T)
            D_f = getDestinationFalse(T)
            Select S = new Select
            setCondition(S, C)
            setValueTrue(S, D[D_t])
            setValueFalse(S, D[D_f])
            append(B, S) valueFrom(P,B,S)
        else                                  ▷ It can only be an unconditional branch
            D_u = getDestination(T) valueFrom(P,B,D[D_u])
        end if
        UnconditionalBranch U = new UnconditionalBranch
        setDestination(U, M)
        append(B, U)
    end for
    Switch S = new Switch
    for all BasicBlock B in D do
        setDestinationIfValue(S, D[B], B)
    end for
    append(M, S)
end function
```

Algorithm 6.12 obfuscateconstants

```
function OBFUSCATECONSTANTS(Module & M)
    ArrayPointer P = new ArrayPointer                ▷ To be able to access the array
    ADDGLOBAL(M, P)
    ConstantList L = new ConstantList          ▷ Constants moved to memory go here
    for all Function F in M do
        CPRNG R = PRNG(F,"obfuscateconstants")
        if not ISNULL(R) then
            for all Instruction I1 in I do
                if ISPHINODE(I1) then
                    for all BasicBlock B2 in GETFROM(I1) do
                        Instruction I2 = GETTERMINATOR(B2)
                        Value V = GETVALUEFROM(P,B2)
                        OBFUSCATEUSE(R, P, L, I2, V)
                    end for
                else
                    for all Value V in I do
                        if CANBEINSTRUCTION(V,I) then
                            OBFUSCATEUSE(R, P, L, I1, V)
                        end if
                    end for
                end if
            end for
        end if
    end for
    Array A = ARRAYFROMLIST(L)
    ADDGLOBAL(M, A)
    SETVALUE(P)GETREFERENCE(A)                                        ▷ Point P to A
end function
```

Algorithm 6.13 obfuscateUse

```
function OBFUSCATEUSE(CPRNG & R, ArrayPointer P, ConstantList & L, Instruction I1, Value & V)
    if ISCONSTANT(V) then
        Instruction I2 = OBFUSCATECONSTANT(R, P, L, I1, V)
        if C ≠V then
            obfuscatedConstants = obfuscatedConstants + 1
            REPLACE(V,I2)
        end if
    end if
end function
```

Algorithm 6.14 obfuscateConstant

function OBFUSCATECONSTANT(CPRNG & R, ArrayPointer P, ConstantList & L, Instruction I_1, Value & V)
    Integer S_V = BITLENGTH(V)
    Integer S_A = BITLENGTH(GETREFERENCEDTYPEP())
    if ISINTEGER(V) **then**
        if $S_V \leq S_A$ **and** WITHPROBABILITY(R, $\frac{1}{2}$) **then**
            return MOVETOARRAY(R, P, L, I_1, V)
        else
            return CREATEOPERATION(R, P, L, I_1, V)
        end if
    end if
    return V
end function

Algorithm 6.15 moveToArray

function MOVETOARRAY(CPRNG & R, ArrayPointer P, ConstantList & L, Instruction I_1, Value & V)
    Probability P_R = GETREOBFUSCATIONPROBABILITY()
    Integer S_V = BITLENGTH(V)
    Integer S_A = BITLENGTH(GETREFERENCEDTYPEP())
    Constant C_1 = SIZE(L)
    APPEND(L,V)
    if WITHPROBABILITY(R, P_R) **then**
        C_1 = OBFUSCATECONSTANT(R, P, L, I_1, C_1)
        reobfuscatedConstants = reobfuscatedConstants + 1
    end if
    Instruction I_2 = CREATELOAD(P)
    INSERTBEFORE(I_1,I_2)
    Instruction I_3 = CREATEGETARRAYADDRESS(I_2,C_1)
    INSERTBEFORE(I_1,I_3)
    Instruction I_4 = CREATELOAD(I_3)
    INSERTBEFORE(I_1,I_4)
    Instruction V_R = I_4
    if $S_V < S_A$ **then**
        Instruction I_5 = CREATETRUNCATE(I_4,BITLENGTH(V))
        INSERTBEFORE(I_1,I_5)
        V_R = I_5
    end if
    return V_R
end function

Algorithm 6.16 createOperation

```
function createOperation(CPRNG & R, ArrayPointer P, ConstantList & L, Instruction I1, Value & V)
    Probability P_R = getReobfuscationProbability()
    Constant C1 = getRandomInteger(R)
    Operation O
    if withProbability(R, P_R/2) then
        C1 = obfuscateConstant(R, P, L, I1, C1)
        reobfuscatedConstants = reobfuscatedConstants + 1
    end if
    Constant C2
    if withProbability(R, 1/3) then
        C2 = V − C1
        O = createAddOperation()
    else if withProbability(R, 1/2) then
        C2 = V + C1
        O = createSubstractOperation()
    else
        C2 = V ⊕ C1
        O = createXorOperation()
    end if
    if withProbability(R, P_R/2) then
        C2 = obfuscateConstant(R, P, L, I1, C2)
        reobfuscatedConstants = reobfuscatedConstants + 1
    end if
    Instruction I2 = createOperationInstruction(O,C2,C1)
    InsertBefore(I1,I2)
    return I2
end function
```

6.3 Polymorphic

The polymorphic transformations do not aim to make the code more difficult to read but different every time it is run, according to the results of a PRNG. This results in smaller penalties for using the transformations but can make the code harder to compare.

6.3.1 bbsplit

This transformation will go over all of the basic blocks of the function and, for each basic block, decide on splitting it for each instruction (except for the PHI nodes and the first non PHI node instruction).

As splitting can alter the basic blocks list, all of the initial basic blocks are stored on a vector, upon which splitting is then run.

The probability of splitting a basic block at each particular point can be adjusted by using the splitprobability parameter. Keep in mind, though, that setting the parameter to one will result in each instruction being split.

The pseudocode for the transformation is provided at algorithm 6.17.

6.3.2 randbb

This transformation applies the basic blocks reordering algorithm to each function. Of greatest importance in this step is keeping the entry block the same.

The pseudocode for the transformation is provided at algorithm 6.18.

6.3.3 randins

This transformation applies the instructions reordering algorithm to each basic block.

The code could be improved upon by separating the PHI node handling function from the more complex handling of normal instructions. Again, has not been done because of the time constraints of the project.

The pseudocode for this transformation is provided at algorithm 6.19.

6.3.4 randfun

This transformation applies the functions reordering algorithm to each module.

Although not necessarily useful for binary patch obfuscation, this transformation was developed because of the aid it provided in code hardening at compilation time.

The pseudocode for the transformation is provided at algorithm 6.23.

6.3.5 randglb

This transformation applies the globals reordering algorithm to each module.

Again, although not of interest for binary patch obfuscation, the transformation was developed for the assistance it provides in code hardening at compilation time.

The pseudocode for this transformation is provided at algorithm 6.24.

6.3.6 swapops

This transformation applies the operands reordering algorithm to each module, which usually results in different registers being allocated on the ensuing assembly code.

The pseudocode for the transformation is provided at algorithm 6.25.

Algorithm 6.17 bbsplit

```
function BBSPLIT(Function & F)
    CPRNG R = PRNG(F,"bbsplit")
    if not ISNULL(R) then
        BasicBlockQueue Q = new BasicBlockQueue
        Probability P_S = GETSPLITPROBABILITY()
        for all BasicBlock B in F do
            APPEND(Q, B)                          ▷ Queue blocks to avoid trouble
        end for
        for all BasicBlock B in Q do
            for all Instruction I in B do
                if not ISPHINODE(I) and not ISFIRSTNONPHI(B,I) then
                    if WITHPROBABILITY(R, P_S) then
                        SPLITAT(I, B)
                    end if
                end if
            end for
        end for
    end if
end function
```

Algorithm 6.18 randbb

```
function randbb(Function & F)
    CPRNG R = prng(F,"randbb")
    if not isNull(R) then
        BasicBlockArray A = new BasicBlockArray
        BasicBlock I = getEntryBlock(F)
        for all BasicBlock B in F do
            if B ≠ I then
                append(A, B)
            end if
        end for
        randomizeOrder(R, A)
        for all BasicBlock B in A do
            if B ≠ I then
                moveAfter(I, B)
                I = B
            end if
        end for
    end if
end function
```

Algorithm 6.19 randins

```
function randins(Function & F)
    CPRNG R = prng(F,"randins")
    if not isNull(R) then
        for all BasicBlock B in F do
            reorderPHINodes(R, B)
            Instruction2InstructionSetMap & M
            InstructionList & L
            M, L = createDependencyMap(B)
            reorderNonPHINodes(R, B, M, L)
        end for
    end if
end function
```

Algorithm 6.20 reorderPHINodes

```
function REORDERPHINODES(CPRNG & R, BasicBlock & B)
    PHINodeArray A = new PHINodeArray
    for all Instruction I in B do
        if ISPHINODE(I) then
            APPEND(A, I)
        end if
    end for
    RANDOMIZEORDER(R, A)
    PHINode I = GETFIRSTPHINODE(B)
    for all PHINode P in A do
        if P ≠ I then
            MOVEBEFORE(I, P)
        else
            I = P
        end if
    end for
end function
```

Algorithm 6.21 createDependencyMap

```
function createDependencyMap(BasicBlock & B)
    Instruction2InstructionSetMap M = new Instruction2InstructionSetMap
    InstructionList L = new InstructionList
    for all Instruction I_1 in B do
        if not isPHINode(I_1) then
            if hasSideEffects(I_1) then
                for all Instruction I_2 in instructionsAfterI_1( )do
                    if hasSideEffects(I_2) or readsMemory(I_2) then
                        append(M[I_1], I_2)
                    end if
                end for
            end if
            if readsMemory(I_1) then
                for all Instruction I_2 in instructionsAfterI_1( )do
                    if hasSideEffects(I_2) then
                        append(M[I_1], I_2)
                    end if
                end for
            end if
            for all Operand O in I_1 do
                Boolean O_I = isInstruction(O)
                Boolean O_A = isAfter(O,I_1)        ▷ operand must be after instruction
                Boolean O_P = not isPHINode(O)
                Boolean O_B = getBasicBlock(O) ≠B
                if O_I and O_A and O_P and O_B then
                    append(M[I_1], O)
                end if
            end for
            if empty(M[I_1]) then
                append(L, I_1)
            end if
        end if
    end for
    return M, L
end function
```

Algorithm 6.22 reorderNonPHINodes

```
function reorderNonPHINodes(CPRNG & R, BasicBlock & B, Instruction2InstructionSetMap & M, InstructionList & L)
    Instruction I1 = getFirstNonPhi(B)
    while not empty(L) do
        Instruction I2 = extractRandomElement(R,L)
        if I2 ≠ I1 then
            moveBefore(I1, I2)                          ▷ move I2 before I1
        else
            I1 = getNext(I1)
        end if
        for all User U in I2 do
            if isInstruction(U) and isOn(M,U) then
                remove(M[U], I2)
                if empty(M[U]) then
                    remove(M[U])
                    append(L, J)
                end if
            end if
        end for
    end while
end function
```

Algorithm 6.23 randfun

```
function randfun(Module & M)
    CPRNG R = prng(M,"randfun")
    if not isNull(R) then
        FunctionArray A = new FunctionArray
        for all Function F in M do
            append(A, F)
        end for
        randomizeOrder(R, A)
        Function I = getFirstFunction(M)
        for all Function F in A do
            if F ≠ I then
                moveAfter(I, F)
                I = F
            end if
        end for
    end if
end function
```

Algorithm 6.24 randglb

```
function randglb(Module & M)
    CPRNG R = prng(M,"randglb")
    if not isNull(R) then
        GlobalArray A = new GlobalArray
        for all Global G in M do
            append(A, G)
        end for
        randomizeOrder(R, A)
        Global I = getFirstGlobal(M)
        for all Global G in A do
            if G ≠ I then
                moveAfter(I, G)
                I = G
            end if
        end for
    end if
end function
```

Algorithm 6.25 swapops

```
function swapops(Function & F)
    CPRNG R = prng(F,"swapops")
    if not isNull(R) then
        for all Instruction I in F do
            Boolean I_C = isConmutative(I)
            Boolean I_B = isBinary(I)
            if I_C and I_B and withProbability(R, P_S) then
                swapOperands(I)
                swappedOperands = swappedOperands + 1
            end if
        end for
    end if
end function
```

7

Evaluation

7.1 Proof: the CPRNG is a good PRNG

As the CTR mode based CPRNG being used is at the heart of this project's transformations, it is important to prove that it follows the properties desirable for any Pseudo-Random Number Generation in order to demonstrate that the use of such a generator is adequate.

In the following subsections, proof is provided that the CPRNG has the properties of determinism, uniformity, and independence. Additionally, its period is calculated.

7.1.1 Determinism

Determinism is given by the fact that no random data is used to generate the key used by CTR mode and that the original IV is the same. Thus, as block ciphers need to be deterministic to allow decryption on the other side, the CPRNG is deterministic.

7.1.2 Uniformity

Given that block ciphers are a one to one mapping of n-bit blocks to n-bit blocks and that the IV is incremented by one each time, the CPRNG will cover all of the 2^n possible inputs (and thus the 2^n possible outputs), generating the largest possible uniform output.

7.1.3 Independence

Since the mapping done by the encryption algorithm is based on the key used, all outputs are independent from each other, as long as the encryption algorithm is a pseudorandom permutation.

7.1.4 Function period

As the counter iterates over the total 2^n states that are possible with its n-bits, and as each input block is mapped to a different and unique ouput block, the period of the CPRNG is exactly 2^n, which should be large enough for any practical use.

7.2 Proof: the CPRNG is secure

In a similar way, because the CTR mode-based CPRNG used for this project can be attacked for the purpose of determining the decisions made during the transformations, it is important to prove that it follows the properties desirable for any Cryptographic Pseudo-Random Number Generator, thus ensuring that the use of such a generator is adequate.

In the following subsections, proof is provided that the CPRNG has the following properties: resistance to next bit tests; impossibility of deriving the function result if the state is known; and, based on this, resistance to the state compromise extension.

7.2.1 Next bit-test resistance

As long as the block encryption algorithm used for the CTR mode is resistant to cryptanalysis, it will be impossible to derive the key used (and thus the state) to predict the next block that will be generated. Inside blocks this property is held, as the cipher is a pseudorandom permutation, and thus no bit presents a visible dependence from the previous one.

7.2.2 Impossibility of knowing the result if only the state is known

One of the problems with the CPRNG is that the seed used for the state (the IV of the CTR mode) is known (and is zero); however, since the attacker has no way of knowing the key (if the obfuscation key is kept secret), it is impossible for him to know which of all the possible blocks will be generated by AES.

7.2.3 State compromise extension resistance

Since the key used in CTR is hidden and is not part of the state (which is only the IV), knowing the value of the IV provides no information, as long as the key is resistant to known plaintext attacks. As a result, given the impossibility of knowing the result if only the state is known, even if the state is known and previous and future states can be derived, it is impossible for the attacker to know the result of the function, thus making the algorithm secure.

7.3 Proof: the key derivation is secure

The first CMAC iteration is performed using a symmetric key encryption algorithm as a hash function in order to summarize the entropy of the obfuscation key string. Since CMAC uses the previous AES outputs to calculate the next one, this effectively results in all of the entropy from the original key being kept and compressed in the resulting tag (with up to the 2^{128} bits possible as output).

The second CMAC iteration uses the resulting key as the key to encrypt a string made of publicly known data (an identifier depending on the function name being available or not, the module name, and the transformation name).

Since the obfuscation key is only used as the key of the CMAC algorithm, it is impossible for the attacker to derive it without actually breaking AES. Additionally, the entropy provided by the key and the input string is effectively summarized by CMAC into a smaller string which can be used as a key for CTR, as proved above.

7.4 Reversing the transformations

During evaluation of the implemented transformations it was discovered that it is possible to reverse each of them. The following sections describe a method for doing so, although this method was not implemented, due to time constraints.

The objective is not to get back to the original code (doing so is most likely impossible without breaking the CPRNG), but to gain a set of transformations that, when applied to the original and the obfuscated assembly, will result in the same **LLVM** IR. (Thus, if the obfuscated code is equal to the one previously provided, it will result in the same IR code.) Furthermore, when the obfuscated assembly is different from the original assembly, the resulting IR code will only be correspondingly different, allowing an attacker to focus only on the vulnerability.

The possibility of reversing the transformations, though, depends on the possibility of transforming the resulting assembly back into **LLVM**'s IR. A way to achieve this is by modeling each instruction of the assembly language into one or more equivalent instructions in **LLVM**'s IR, and then transforming register accesses into memory reads and writes (which could be optimized later).

Currently, no known library is able to accomplish this, but it is reasonable to predict that one may be developed in the future.

7.4.1 Defining a global ordering of values

Values can be constants or instruction results. Defining their ordering is important, as it allows the deobfuscating program to define how to order the "contents" of an instruction (i.e., the operands). The value ordering given the instructions I and J is defined at algorithm 7.1.

7.4.2 Defining a global ordering of instructions

This is the core of reversing most of the code reordering transformations, as when instructions can be ordered an ordering can also be created for the contents of basic blocks and functions.

The instruction ordering given the instructions I and J in the same basic block is defined at algorithm 7.2.

7.4.3 Reversing the randfun and randglb transformations

These can be reversed quite trivially by reordering globals and functions alphabetically or, when anonymous or with the same name, by their contents' values.

7.4.4 Reversing the swapops transformation

This can be reversed (once an ordering for values is defined) by ordering the operands of the instruction accordingly, if the instruction is a candidate for operand swapping.

7.4.5 Reversing the randins transformation

This transformation can be reversed by ordering the instructions according to the global ordering in the basic block. At times, two instructions with exactly the same contents may be found. If this happens, the instructions may be merged, resolving the conflict.

7.4.6 Reversing the obfuscateconstants transformation

To reverse this transformation, the only thing that needs to be done is to pass a constant calculation transformation, which will replace instructions by the constant values they calculate and remove any unused global variables.

7.4.7 Reversing the flattencontrol transformation

To reverse this transformation, find the PHI node that chooses the destination of the main basic block switch depending on the basic block which jumped to it. With this, the unconditional jumps cand be replaced by the node chosen on the main basic block, or, when using selection by a conditional basic block, by conditional the jumps depending on the select condition valhe and the node that will be chosen in the main basic block.

Afterwards, move the PHI nodes to the first basic block where they are used, according to the CFG and a liveness analysis, and delete the main basic block.

Finally, apply the Unify Function Exit Nodes transformation to ensure both flow graphs are equal.

7.4.8 Reversing the bbsplit transformation

To reverse this transformation, simply merge any two basic blocks, BB1 and BB2, where BB1 has an unconditional jump to BB2, and BB2 has only BB1 as a predecessor.

7.4.9 Reversing the randbb transformation

This transformation can be reversed through ordering the basic blocks by traversing the CFG using breadth first search and choosing the basic blocks (when two or more are available at the same level) according to the order in which their parents where chosen, or, when the same parents are there, according to the contents of the basic block itself. If the contents are the same, then the basic blocks may be merged, instead.

Since the contents may be different when reversed, this ordering may not result in the same code on both sides, when the basic block contents are not the same. An optimization pass can be used, though, to reduce these differences.

Algorithm 7.1 Global ordering of values

```
function valueOrdering(I, J)
    if isConstant(I) and isConstant(J) then
        return I < J                                  ▷ Normal ordering
    else if isConstant(I) and not isConstant(J) then
        return true                                   ▷ I goes before J
    else if not isConstant(I) and isConstant(J) then
        return false                                  ▷ J goes before I
    else
        return instructionOrdering(I,J)               ▷ Use the global ordering instead
    end if
end function
```

Algorithm 7.2 Global ordering of instructions

```
function instructionOrdering(I, J)
    if depends(I, J) or depends(J, I) then
        return precedes(I, J)        ▷ Order by precedence if there are dependencies
    else if operandCount(I) < operandCount(J) then
        return true                                   ▷ Use operand count
    else if operandCount(I) > operandCount(J) then
        return false                                  ▷ Use operand count
    else if opCode(I) < opCode(J) then
        return true                                   ▷ Use opcode ordering
    else if opCode(I) > opCode(J) then
        return false                                  ▷ Use opcode ordering
    else
        for all IOP, JOP in pairs(getOperands(I), getOperands(J)) do
            if IOP ≠ JOP then
                return valueOrdering(IOP, JOP)   ▷ Order according to operands
            end if
        end for
    end if
end function
```

7.5 Binary patch obfuscation technique evaluation

THE proposed technique consists of obfuscating some of the functions of the code being patched along with the patched function, choosing these extra functions at random.

It is easy to see that if the attacker has no knowledge of which function was modified and cannot reverse the transformations, he will need to analyze the mean of half of the added functions before finding the changed functions, and analyze all of the added functions before he can be certain no other functions are unmodified.

Additionally, if a function only introduces the security fix, then the probability of the reverse engineer finding it after x attempts is inversely proportional to the number of added functions and directly proportional to the number of attempts.

Sadly, an experiment to check how efficient the above obfuscation techniques are could not be run, but, in theory, they should be as efficient as the original techniques they are based upon. This lack of an experiment has made it impossible to measure the amount of extra time that is required to analyze obfuscated functions.

8

Conclusions

We have developed a set of transformations which allow the focused obfuscation of functions so that only these will be different on the resulting patch. Such transformations have also been implemented into **LLVM**.

In the evaluation section, proof was provided that, given enough interest, the proposed polymorphism transformations and obfuscation transformations can be reversed or, when reversal is not possible, a similar transformation can be applied to both codes to attain a minimal set of differences between the original and the modified code. A proof that if the passes cannot be reversed, the difficulty of finding the security fix increases proportionally to the number of extra obfuscated functions is also provided.

The results of the evaluation can be considered an example of the never-ending war between researchers trying to elaborate better obfuscation techniques, and attackers trying to reverse them. This situation will end either when a technique which cannot be reversed is developed or when newer techniques cannot be created by developers. Sadly, it currently appears as if the second possibility is more likely to happen than the first, as the ways in which programs can be obfuscated are limited and human thinking can adapt to read obfuscated code.

9

Future development

Given the time constraints of this project, many possible avenues could not be explored in this project. The first task that should be performed in the future is to improve the code quality of the transformations so that they can be pushed onto the upstream **LLVM**.

In the constant obfuscation transformation, improvement can be made to the constant memory fetch obfuscation by using Wang's aliasing method and randomly reordering the constant array. Another possible improvement to consider is that the Register Swap could instead be performed over the resulting assembly by remapping registers (which sadly are API-dependent). Also, A study of the efficiency of the transformations should be run, although some preliminary tests hint of roughly a 6x slowdown. Finally, other obfuscation techniques could be applied to render the resulting patches more difficult for attackers to analyze.

As part of the development of this project, it was also discovered that some obfuscation techniques can be used to harden the resulting binaries against certain attacks, with apparently negligible impact. In-depth research on this topic will be performed in the near future.

Bibliography

[1] Arini Balakrishnan and Chloe Schulze. *Code Obfuscation Literature Survey*. 2005. URL: http://pages.cs.wisc.edu/~arinib/writeup.pdf.

[2] Boaz Barak et al. "On the (im)possibility of obfuscating programs". In: *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 1–18. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.8717&rep=rep1&type=pdf.

[3] Eli Bendersky. *Life of an instruction in LLVM*. Nov. 2012. URL: http://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm/.

[4] Jean-Marie Borello and Ludovic Mé. "Code obfuscation techniques for metamorphic viruses". English. In: *Journal in Computer Virology* 4.3 (2008), pp. 211–220. ISSN: 1772-9890. DOI: 10.1007/s11416-008-0084-2. URL: http://dx.doi.org/10.1007/s11416-008-0084-2.

[5] Frederick P. Brooks Jr. "No Silver Bullet Essence and Accidents of Software Engineering". In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532. URL: http://dx.doi.org/10.1109/MC.1987.1663532.

[6] Chih-Fan Chen et al. *CONFUSE: LLVM-based Code Obfuscation*. May 2013. URL: http://www.cs.columbia.edu/~aho/cs4115_Spring-2013/lectures/13-05-16_Team11_Confuse_Paper.pdf.

[7] S. Garg et al. "Candidate Indistinguishability Obfuscation and Functional Encryption for all Circuits". In: *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. Oct. 2013, pp. 40–49. DOI: 10.1109/FOCS.2013.13. URL: http://eprint.iacr.org/2013/451.pdf.

[8] Anne Gille-Genest. *Pseudo-Random Numbers Generators*. Mar. 2012. URL: https://quanto.inria.fr/pdf_html/mc_random_doc/#x1-40001.2.

[9] Brian Gladman. *AES and Combined Encryption/Authentication Modes*. 2014. URL: http://gladman.plushost.co.uk/oldsite/AES/index.php.

[10] Sebastian Hack and Gerhard Goos. "Optimal Register Allocation for SSA-form Programs in Polynomial Time". In: *Inf. Process. Lett.* 98.4 (May 2006), pp. 150–155. ISSN: 0020-0190. DOI: 10.1016/j.ipl.2006.01.008. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=4E2D4510A92E0FD0AB2775F446362B18?doi=10.1.1.204.2844&rep=rep1&type=pdf.

[11] Francisco Blas Izquierdo Riera. "Pint, herramienta de simulación basada en trazas Pin". MA thesis. Polytechnical University of Valencia, Dec. 2012. URL: http://hdl.handle.net/10251/18304.

[12] Francisco Blas Izquierdo Riera. *The SIV mode of operation result in data leakage with small messages (<= blocksize) when the authentication part of the key is discovered and how to get data from CMAC*. June 2011. URL: http://seclists.org/fulldisclosure/2011/Jun/382.

[13] Pascal Junod. *Obfuscator-LLVM*. 2013. URL: https://github.com/obfuscator-llvm/obfuscator/wiki.

[14] Pascal Junod. *Obfuscator reloaded*. Nov. 2012. URL: http://crypto.junod.info/asfws12_talk.pdf.

[15] Cullen Linn and Saumya Debray. "Obfuscation of Executable Code to Improve Resistance to Static Disassembly". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. Washington D.C., USA: ACM, 2003, pp. 290–299. ISBN: 1-58113-738-9. DOI: 10.1145/948109.948149. URL: http://doi.acm.org/10.1145/948109.948149.

[16] LLVM Project. *CommandLine 2.0 Library Manual*. Mar. 2014. URL: http://llvm.org/docs/CommandLine.html.

[17] LLVM Project. *LLVM Language Reference Manual*. 2014. URL: http://llvm.org/docs/LangRef.html.

[18] LLVM Project. *Writing an LLVM Pass*. 2014. URL: http://llvm.org/docs/WritingAnLLVMPass.html.

[19] Matias Madou, Ludo Van Put, and Koen De Bosschere. "LOCO: An Interactive Code (De)Obfuscation Tool". In: *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. PEPM '06. Charleston, South Carolina: ACM, 2006, pp. 140–144. ISBN: 1-59593-196-1. DOI: 10.1145/1111542.1111566. URL: http://doi.acm.org/10.1145/1111542.1111566.

[20] Matias Madou et al. *Code (De)Obfuscation*. June 2005. URL: http://escher.elis.ugent.be/publ/Edocs/DOC/P105_076.pdf.

[21] Matias Madou et al. "On the Effectiveness of Source Code Transformations for Binary Obfuscation". In: *Proc. of the International Conference on Software Engineering Research and Practice (SERP06)*. 2006. URL: https://biblio.ugent.be/publication/374659/file/496495.pdf.

[22] Jeongwook Oh. “Fight Against 1-day Exploits: Diffing Binaries vs Anti-diffing Binaries”. In: *Black Hat USA 2009 //Media Archives*. Las Vegas, NE, USA, 2009, p. 1. URL: http://www.blackhat.com/presentations/bh-usa-09/OH/BHUSA09-Oh-DiffingBinaries-PAPER.pdf.

[23] Axel ”0vercl0k” Souchet. *Obfuscation of steel: meet my Kryptonite*. July 2013. URL: http://download.tuxfamily.org/overclokblog/Obfuscation%20of%20steel%3a%20meet%20my%20Kryptonite/0vercl0k_Obfuscation_of_steel_meet_kryptonite.pdf.

[24] StatCounter. *Top 8 Operating Systems from Feb 2013 to Jan 2014*. Feb. 2014. URL: http://gs.statcounter.com/#all-os-ww-monthly-201302-201401.

[25] StatCounter. *Top 9 Browsers from Feb 2013 to Jan 2014*. Feb. 2014. URL: http://gs.statcounter.com/#all-browser-ww-monthly-201302-201401.

[26] Teja Tamboli. “Metamorphic Code Generation from LLVM IR Bytecode”. MA thesis. San José State University, 2013. URL: http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1305&context=etd_projects.

[27] The Clang Team. *“Clang” CFE Internals Manual*. 2014. URL: http://clang.llvm.org/docs/InternalsManual.html#how-to-add-an-attribute.

[28] S.K. Udupa, S.K. Debray, and M. Madou. “Deobfuscation: reverse engineering obfuscated code”. In: *Reverse Engineering, 12th Working Conference on*. Nov. 2005, 10 pp.–. DOI: 10.1109/WCRE.2005.13. URL: http://dx.doi.org/10.1109/WCRE.2005.13.

[29] University of Illinois at Urbana-Champaign. *LLVM API Documentation: llvm::FunctionPass Class Reference*. Mar. 2014. URL: http://llvm.org/docs/doxygen/html/classllvm_1_1FunctionPass.html.

[30] University of Illinois at Urbana-Champaign. *LLVM API Documentation: llvm::ModulePass Class Reference*. Mar. 2014. URL: http://llvm.org/docs/doxygen/html/classllvm_1_1ModulePass.html.

[31] University of Illinois at Urbana-Champaign. *LLVM API Documentation: llvm::RegisterPass$<$ passName $>$ Struct Template Reference*. Mar. 2014. URL: http://llvm.org/docs/doxygen/html/structllvm_1_1RegisterPass.html.

[32] Chenxi Wang. “A Security Architecture for Survivability Mechanisms”. PhD thesis. Charlottesville, VA, USA: Faculty of the School of Engineering and Applied Science at the University of Virginia, 2001. ISBN: 0-493-08929-2. URL: http://www.cs.virginia.edu/~jck/publications/wangthesis.pdf.

[33] Gregory Wroblewski. “General Method of Program Code Obfuscation”. PhD thesis. Wrocław University of Technology, 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.9052&rep=rep1&type=pdf.

[34] Ilsun You and Kangbin Yim. "Malware Obfuscation Techniques: A Brief Survey". In: *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. Nov. 2010, pp. 297–300. DOI: 10.1109/BWCCA.2010.85. URL: http://dx.doi.org/10.1109/BWCCA.2010.85.

[35] Dan Zimmerman. *Got AES Performance?* Oct. 2010. URL: http://software.intel.com/en-us/blogs/2010/10/14/got-aes-performance.

A

Using the tools

THE project transformations are compiled into their own library at `lib/Obf.so` and need to be loaded explicitly when using `opt` through the `--load` switch. This is done mainly to keep the code isolated from the rest of the tools, making it easier to integrate.

An example of how to load the library is shown at code listing A.1.

Code listing A.1 Loading the obfuscation library

```
$ opt --load ./Release+Asserts/lib/Obf.so
```

`opt` takes as input an **LLVM** IR program and outputs another, transformed, one. The transformations are applied in the order given. The following switches will add a pass with each of the different transformations:

`-addmodulekey` Enables the transformation for adding a module key.

`-bbsplit` Enables the bbsplit transformation which randomly splits basic blocks.

`-flattencontrol` Enables the control flattening transformation which will flatten the marked functions.

`-obfuscateconstants` Enables the constant obfuscation transformation.

`-propagatemodulekey` Enables the transformation for the module key propagation to the module functions.

`-randbb` Enables the transformation for randomly reordering basic blocks.

`-randfun` Enables the transformation for randomly reordering functions.

`-randglb` Enables the transformation for randomly reordering globals.

-randins Enables the transformation for performing the dependence-based random reordering of instructions.

-swapops Enables the transformation for randomly swapping instruction operands.

Additionally, some of the transformations have a set of tunable parameters, which can be modified by using the following flags:

-modulekey *string* Defines the key inserted in the module by -addmodulekey.

-splitprobability *probability* Specifies the probability of splitting the basic block at each instruction for -bbsplit.

-reobfuscationprobability *probability* Specifies the probability of reobfuscating a constant.

The order in which these transformations are run can affect the resulting code and the effectiveness of the transformations. Thus, the following order is recommended:

1. Any optimization transformations
2. The -addmodulekey transformation
3. The -propagatemodulekey transformation
4. The -bbsplit transformation
5. The -flattencontrol transformation
6. The -obfuscateconstants transformation
7. The -randins -randbb -randfun -randglb and -swapops transformations

An example of a complete call to opt can be found at code listing A.2.

Code listing A.2 Using opt to obfuscate **LLVM** code

```
$ opt --load ./Release+Asserts/lib/Obf.so -addmodulekey
    -modulekey "Example key" -propagatemodulekey -bbsplit
    -splitprobability 1/8 -flattencontrol -obfuscateconstants
    -reobfuscationprobability 1/5 -randins -randbb -randfun
    -randglb -swapops < input.bc > output.bc
```

In order to work with clang, the -emit-llvm option and an extra call to clang for linking must be added. The line at code listing A.3 contains the procedure to compile and link a c file.

At code listing A.3, a pipeline from clang, to opt, and back to clang is generated.

The first call to clang compiles the provided C file, runs the level 3 standard optimizations with -O3, and stops before linking with -c. It then emits the **LLVM** bytecode with -emit-llvm and outputs it to the standard output with -o -.

Code listing A.3 Compiling an obfuscated binary using clang and opt

```
$ clang file.c -O3 -c -emit-llvm -o - | opt --load
    ./Release+Asserts/lib/Obf.so -addmodulekey -modulekey "Example␣
    key" -propagatemodulekey -bbsplit -splitprobability 1/8
    -flattencontrol -obfuscateconstants -reobfuscationprobability
    1/5 -randins -randbb -randfun -randglb -swapops | clang -x ir -
```

The call to `opt` parses the generated bytecode, as explained above.

Finally, the last call to clang uses `-x ir` to specify that the input contains **LLVM**'s IR (in bytecode form) and a single `-`, in order to make clang take input from the standard input. This will link and compile the IR code generated by `opt` into the `a.out` file. If object code is desired, then the `-c` option can be used, in a similar way. To specify the desired output file, the `-o` option can be used.

B

Popularization

B.1 Programming computers

LIKE many other digital systems, computers work in binary, which is a language where two clearly different values exist. These values are usually referred as 0 and 1, and each of them is considered a bit.

As these values by themselves allow for only two possible states, they can be grouped to provide a larger array of possible values. For example, if two values are grouped, the following four states can be obtained: 00, 01, 10 and 11. In a similar way, three values yield eight different states and, in general, n values produce 2^n states.

By themselves, these groups of values set to 0 or 1 are meaningless, but it is possible to use them to encode data, such as the colors of an image or the amount of money that you have in your bank account. This is done by providing a meaning to each of the bits in the group, so that each of the two possible values will affect the meaning of the group in one way or another.

You may also be aware that computers are programmable. This means that they use some instructions to know what action they need to perform with the groups of bits they utilize. These instructions are also encoded using groups of ones and zeros, so that the computer knows, for example, that it needs to grab a value from a particular place or add the values it can find in two places and put the result in a third place. The meaning of these bit sequences is heavily dependent on the computer type, as different computer designs interpret bits differently.

B.1.1 Assembly

When humans want to give orders or transmit information to each other, they do not say "01110010101", but rather use words like "bring me water". As a result of this, it is difficult for humans to understand or speak directly in binary.

To fix this problem, assembly languages were created. An assembly language is a compromise between the high level languages used by humans and the binary languages used by machines. For example, on an Intel™ processor (like the one on most desktops) the code "0000000011000011" means "add the value of the bits in the locations al and bl and store the result in the location al", which would be written in assembly as "add %al,%bl".

As stated above, though, machines have no way of understanding that "add %al,%bl" is equivalent to "0000000011000011" without a special translation program. This program is called an assembler. Similarly, some situations require a program to decode "0000000011000011" into "add %al,%bl". Such a program is known as a disassembler.

Assembly languages usually allow for some small levels of abstraction, which permits humans to more easily understand what is being coded. For example, these languages may allow comments to be added, explaining what different parts of the code do or adding descriptive names to different locations where information can be stored for processing. When converting the program into a set of instructions written in binary (which the machine can understand), this information is discarded, as it is unnecessary for the machine (and in many cases it cannot be encoded anyways). As a result, when the disassembler converts the binary instructions back into assembly language, this information will be missing, making the code more difficult to understand.

The main advantage of assembly is that it allows the programmer to use a language which is easier for them to understand whilst still exposing all of the capacities of the machine. In exchange, however, some time needs to be invested in translating the program into the language that the machine understands, although this only needs to be done once. As mappings can be created both ways, it is also possible to convert the original binary code back into an assembler instruction, which qualified individuals can then more easily read.

The main disadvantages of this process is that assembly is still difficult for humans to understand (as they need to imagine the machine performing the instructions in order to visualize what the code actually does), and each machine uses a different assembly language, as they each have different characteristics to expose.

B.1.2 Programming languages

In order to overcome the disadvantages of assembly languages, programming languages were created.

A programming language is a construct which abstracts the details of the machine (allowing the same code to be used on machines with different features) and attempts to provide an interface which is easier for humans to work with. The costs of using programming languages are a greater processing time and the necessity of writing more complex programs which are capable of converting programs written using these languages into binary programs that the machine can understand.

There are multiple paradigms which are differentiated mostly by the way in which they allow the programmer to model the program.

As more abstraction is added, the resources provided by the computer are used less efficiently, but the program is more easily transferable to other computers. Additionally, a simpler interface is provided, which allows the programmer to model the program in a language closer to that of the problem he is trying to solve. The alternate also applies, as more and more details of the machine are provided by the language.

B.2 Compilers

As stated above, machines utilize binary languages which tell them exactly what to do to solve a problem. Thus, a program capable of converting the instructions provided in more abstract programming languages into a binary program is needed. This program is known as a compiler.

In general, compilers do not perform this transformation into machine language directly, as the result would be very complex programs that would execute everything at the same time. Instead, they go over a set of stages which convert more abstract languages into either the same language or a less abstract one. This new program will keep the same meaning as the original.

For example, a program compiled using the **LLVM** suite would first be converted into an assembly like language that hides most of the limitations of real machines, then optimized into a faster program in this same language, converted into a representation where operands indicate their operators, optimized again into a faster program using this representation, and rewritten again so that the final representation matches the limits imposed by the destination machine. Finally, this representation would be converted into assembly language and passed to an assembler that converts that program into binary language.

During the compilation process, compilers, in a way similar to assemblers, discard the information that is not needed by the machine to understand the program. This, along with the fact that there is no direct mapping from the machine instructions to the original abstract representation, makes it more difficult to recover the original program (minus the discarded information) from the binary one provided. Despite these limitations, some programs are able to recognize patterns on the code generated for different structures by compilers. These kind of programs are called decompilers.

B.3 Antireverse engineering

Reverse engineering is the process of taking apart the different pieces that make a product in order to understand how the product works. In a similar way, when applied to a programming context, it is the process of analyzing the machine code contained in a program in order to understand what it does and how it does it.

Reverse engineering has many uses: understanding how a program works, understanding the results it produces to interpret them or generate equivalent results with your own program, modifying the program to override code in order to enforce license restrictions, or even detecting flaws in the program that can be exploited.

Because of the possibility of performing reverse engineering, programmers try to make it difficult for others to understand the machine code produced after compilation. There are different ways of doing this.

One way of making programs harder to reverse engineer consists on removing any unnecessary, human-understandable information which could be contained in the program. This process is generally known as stripping, as the program is "stripped" to the minimum necessary to be executed by the machine.

Another method is to thwart the efforts of decompilers or disassemblers by exploiting some properties of the machine code. Regardless, neither this nor the previous technique will stop people who can understand machine code, and thus the program.

A final method of making the resulting machine code more difficult for humans to understand is the process known as obfuscation. This usually comes with a price, as it may result in larger and slower programs. Also, as shown by [2], some programs cannot be obfuscated.

B.3.1 Obfuscation techniques

As interest in the area of obfuscation grew, more and more techniques to make programs difficult to understand were developed. Similarly, more and more effective methods of reversing obfuscations were created. This has resulted in an arms race, wherein one group develops stronger techniques and the other stronger methods to override them.

Some of these techniques focus simply on making the program different from the original one. These techniques are called polymorphism techniques, as they morph the program into different shapes.

Other techniques try to modify the structure of the program to make the resulting code more difficult to read.

B.3.1.1 Control flattening

Some instructions tell the computer to continue executing instructions which are not next in order. These instructions can be executed only when certain conditions are met which allows for the creation of loops that will repeat the same sequence of instructions either forever, or until a condition is met.

Control flattening works by replacing all of these instructions by a jump to the same sequence of instructions. This sequence will then jump to the originally intended destination based on the information passed before the jump.

To explain the results of such an operation, this recipe will be obfuscated:

1. Put 4 eggs in an empty dish.
2. Add ½ glass of oil to that dish.
3. Put 500 grams of flour in an empty bowl.
4. Add 1 glass of water to the bowl.
5. Add a spoonful of yeast to the bowl.
6. Add the contents of the dish to the bowl.
7. Knead the mix.
8. If the mix is not a consistent dough, then repeat step 7.
9. If the dough has not risen, repeat step 9.
10. Start the oven at a temperature of 180º.
11. If the oven is not at 180º, repeat step 11.
12. Put the dough in the oven.
13. If the bread is not baked, repeat step 13.
14. Remove the bread from the oven.
15. Turn off the oven.
16. You are done.

As is apparent, this recipe is merely a set of simple steps to bake bread using an oven. A computer running a program operates is similar to a human following the steps of a recipe. Now, if the recipe was obfuscated using control flattening, it would look like this:

1. Put 4 eggs in an empty dish.
2. Add ½ glass of oil to that dish.
3. Put 500 grams of flour in an empty bowl.
4. Add 1 glass of water to the bowl.

5. Add a spoonful of yeast to the bowl.
6. Add the contents of the dish to the bowl.
7. Knead the mix.
8. If the mix is not a consistent dough, then write 1 on a paper. Otherwise write 2.
9. Go to step 21.
10. If the dough has not risen, then write 3 on a paper. Otherwise write 4.
11. Go to step 21.
12. Start the oven at a temperature of 180º.
13. If the oven is not at 180º, then write 5 on a paper. Otherwise write 6.
14. Go to step 21.
15. Put the dough in the oven.
16. If the bread is not baked, then write 7 on a paper. Otherwise write 8.
17. Go to step 21.
18. Remove the bread from the oven.
19. Turn off the oven.
20. You are done.
21. If the paper says 1 go to step 7.
22. If the paper says 2 go to step 10.
23. If the paper says 3 go to step 10.
24. If the paper says 4 go to step 12.
25. If the paper says 5 go to step 13.
26. If the paper says 6 go to step 15.
27. If the paper says 7 go to step 16.
28. If the paper says 8 go to step 18.

The result of this transformation is that it is more difficult for a reverse engineer to understand how instructions flow inside the program, as they will first see that all of the jumps go to the same place, and from there to the other instructions.

B.3.1.2 Constant obfuscation

Many programs need constant values to work. As an example, if you want to calculate the price of a product including a fixed amount of taxes, you would need to know the amount of tax in order to add it to the original price. The idea behind constant obfuscation is to make these values harder to find.

For example, imagine that you make a program which will add 4 to the value received as input. You could do this simply by adding 4, or by adding the result of $\frac{(2-1+5)\times 2}{3}$. The second option is obviously harder to understand. Similarly, instead of directly adding 4, you could add the result of fetching information from a particular memory address that you know will return 4.

Using the previous recipe as an example, such a transformation appears as follows:

1. Write ½ on a paper.
2. Put $\frac{(2-1+5)\times 2}{3}$ eggs in an empty dish.
3. Add the amount on the paper glass of oil to that dish.
4. Put $4000 - 7 \times 500$ grams of flour in an empty bowl.
5. Add $\frac{8}{2^3}$ glass of water to the bowl.
6. Add $\frac{3\times 12}{24+6\times 2}$ spoonful of yeast to the bowl.
7. Add the contents of the dish to the bowl.
8. Knead the mix.
9. If the mix is not a consistent dough, then repeat step 8.
10. If the dough has not risen, repeat step 10.
11. Start the oven with a temperature of 180º.
12. If the oven is not at 180º, repeat step 12.
13. Put the dough in the oven.
14. If the bread is not baked, repeat step 14.
15. Remove the bread from the oven.
16. Turn off the oven.
17. You are done.

This technique aims to make values which are constant during the execution of the program more difficult to read, thus making it harder to find these points and use them as references to understand how the program works.

B.3.1.3 Register swap

In computers, some places where binary information is stored have special meanings, such as the place where the next instruction to be executed can be found or the color that needs to be put in a particular place of your screen. The majority of these locations, though, have no meaning other than the one given by the program being run.

The technique known as register swapping randomly alters the meaning of pairs of these otherwise meaningless locations within the program, resulting in a different program.

Using the previous recipe as an example, the result would be:

1. Put 4 eggs in an empty bowl.
2. Add ½ glass of oil to the bowl.
3. Put 500 grams flour in an empty dish.
4. Add 1 glass of water to the dish.
5. Add a spoonful of yeast to the dish.
6. Add the contents of the bowl to the dish.
7. Knead the mix.
8. If the mix is not a consistent dough, then repeat step 7.
9. If the dough has not risen, repeat step 9.
10. Start the oven with a temperature of 180º.
11. If the oven is not at 180º, repeat step 11.
12. Put the dough in the oven.
13. If the bread is not baked, repeat step 13.
14. Remove the bread from the oven.
15. Turn off the oven.
16. You are done.

As you can see, the ingredients that would be in the dish were changed with those in the bowl. The change may not seem meaningful at first, but a careful check reveals that 6 out of the 16 instructions of the recipe were altered.

The result of this technique is code that is different every time it is compiled, thus making it more difficult for a reverse engineer to uncover the differences.

B.3.1.4 Instruction Reordering

The last of the applied techniques consists of randomly reordering the instructions a program executes, if they have no dependencies.

Using the same recipe from above, this transformation would appear as follows:

1. Add ½ glass of oil to an empty dish.
2. Put 4 eggs in the dish.
3. Add 1 glass of water to an empty bowl.

4. Add the contents of the dish to the bowl.
5. Add a spoonful of yeast to the bowl.
6. Put 500 grams flour in the bowl.
7. Knead the mix.
8. If the mix is not a consistent dough, then repeat step 7.
9. If the dough has not risen, repeat step 9.
10. Start the oven with a temperature of 180º.
11. If If the oven is not at 180º, repeat step 11.
12. Put the dough in the oven.
13. If the bread is not baked, repeat step 13.
14. Turn off the oven.
15. Remove the bread from the oven.
16. You are done.

A similar process can be performed by reordering the sequences of instructions which will always be executed in the same order. The recipe could then be modified to look like this:

1. Go to step 8.
2. Remove the bread from the oven.
3. Turn off the oven.
4. You are done.
5. Start the oven with a temperature of 180º.
6. If the oven is not at 180º, repeat step 6.
7. Go to step 15.
8. Put 4 eggs in an empty dish.
9. Add ½ glass of oil to the dish.
10. Put 500 grams flour in an empty bowl.
11. Add 1 glass of water to the bowl.
12. Add a spoonful of yeast to the bowl.
13. Add the contents of the dish to the bowl.
14. Go to step 20.
15. Put the dough in the oven.
16. If the bread is not baked, repeat step 16.

17. Go to step 2.
18. If the dough has not risen, repeat step 18.
19. Go to step 5.
20. Knead the mix.
21. If the mix is not a consistent dough, then repeat 20.
22. Go to step 18.

The result of this technique is a program where the order of the elements is changed every time, thus making it more difficult to find differences from the original program.

B.3.2 Focused obfuscation

To allow for some balance between the penalties introduced by obfuscation and the benefits it provides by making programs harder to reverse engineer, obfuscation techniques are focused in only some parts of the program.

This provides certain benefits. First, only the obfuscated parts of the program will change. (Thus, less changes are sent to the user when he needs to update the program to a new version). Second, only the obfuscated parts will receive the penalties introduced by the obfuscation techniques (thus reducing the total impact). Finally, by using a secret number to define how the techniques will be applied to different parts of the program (or not applied at all), it is possible to always generate the same program, making updating previously obfuscated programs easier, as the parts using the same number will remain the same.

The main drawback of this technique is that by focusing the obfuscation on particular parts of the program, the attacker can concentrate their efforts on those sections, as they will expect relevant aspects of the code to be there. This problem can be solved by choosing many irrelevant parts of the program to also be obfuscated.

B.3.3 Compiler-level obfuscation

It is possible to create compilers which will obfuscate programs as they process them. These compilers provide certain advantages.

Firstly, such compilation simplifies the process for the user of the compiler, as they simply need to tell the compiler to obfuscate the program.

Additionally, the obfuscation technique can be applied in a machine independent language, so the obfuscation code can be used across machines using different designs.

Fianlly, this compiler simplifies the process of focusing the obfuscation techniques, as the user can simply mark the structures that should be obfuscated.

The problem is that some obfuscation techniques rely on features which are not modeled by the language used when the obfuscation is applied. As a result, these techniques cannot be implemented using that language, although they may be implemented in one which is closer to the language that the computer understands.

B.4 Deobfuscation

It is possible to reverse obfuscation techniques that have been implemented in a more or less automated manner. Even though tools to do this do not yet exist, they may be developed in the near future, as interest in their creation increases. Because of this, care should be taken when looking for new developments in the field of research before using one technique or another, as they may only introduce performance penalties without providing any benefit.

B.4.1 Control unflattening

The idea behind this technique is to find the code block where the real control flow of the program is decided, and then to move these decisions to the jumps to that code block. As the decisions are constant values, it is reasonably possible to do this automatically.

B.4.2 Constant deobfuscation

The idea used in this case is that, as constants will keep the same value during the whole execution, they can be calculated once found, thus returning the original values instead of the obfuscated ones.

B.4.3 Register swap

If you define a way to order the places where information is stored based on the instruction being executed and the dependencies amongst instructions, you can then order all of the possible places where information can be stored and assign them based on the ordering you defined. By doing this on the original and the patched program you should end with barely similar programs.

B.4.4 Instruction Reordering

By using the previously defined ordering, you can also order the instructions in both the original and the modified program, thus reducing the amount of changes between them.

B.5 Program updates

Developers may have many reasons for creating new versions of programs and sending them to the users of that program. In some cases, they may have added new features to the program, while in others they may have fixed errors that were discovered. In some situations, these errors are reasonably harmless, but when they can be used by a third party to make the program behave in an undesired way they are considered security vulnerabilities.

Programmers can simply send the updated version of the program to its users, but this is generally inefficient, as most of the program will remain the same, and can even be problematic if the program is quite large. To prevent this, instructions explaining how to create the new version of the program from the older one are sent instead. These instructions are usually known as a patch, and they can be applied automatically by a special program.

Using patches comes with some drawbacks. First of all, the new version of the program has to be similar enough to the old version for the patch to be smaller than the final program. For example, if polymorphic techniques are applied, the whole program would change, making this process inefficient.

Another drawback is that an attacker can focus on the changes introduced by the patch to discover what security vulnerabilities were fixed and once found, exploit them on the users who have not yet updated the program. This kind of attack is known as 1-day exploit, as it is done after the updated program is released.

B.6 Practical example

Suppose a developer is reporting a security vulnerability in a program they developed. He fixes the issue and prepares a patch. In order to prevent an attacker from simply looking at the changes introduced to patch the program, the developer could obfuscate the section of the program that needed to be updated.

As stated above, the attacker can still focus their efforts on the parts of the program that were modified, even if obfuscated, so the developer then decides to also obfuscate and modify also other parts of the program. Thus,the whole program is not changed, but the attacker now needs to read a mean of half of the changes introduced before he can find the one which fixes the vulnerability.

Obviously, these techniques do not prevent the attacker from eventually finding the error being patched and potentially exploiting it in the computers of those who have not upgraded the program, but they can delay the attacker and, by doing so, allow more users to update the program before the attacker can abuse the vulnerability.

As you probably have inferred from the previous chapters, the use of obfuscation techniques can make the program less efficient. Thus, the developer should release a non-obfuscated patch after enough time for the users to upgrade the program has passed.

C

Source code

C.1 Patches to LLVM

```
--- lib/Transforms/Makefile     (revision 192535)
+++ lib/Transforms/Makefile     (working copy)
@@ -9,6 +9,6 @@

 LEVEL = ../..
-PARALLEL_DIRS = Utils Instrumentation Scalar InstCombine IPO
↪Vectorize Hello ObjCARC
+PARALLEL_DIRS = Utils Instrumentation Scalar InstCombine IPO
↪Vectorize Hello ObjCARC Obf

 include $(LEVEL)/Makefile.config
```

C.2 Patches to Clang

```
--- tools/clang/include/clang/Basic/Attr.td     (revision 192535)
+++ tools/clang/include/clang/Basic/Attr.td     (working copy)
@@ -565,6 +565,12 @@
   let Subjects = [ParmVar];
 }
 
+def ObfKey : InheritableAttr {
+  let Spellings = [GNU<"obfkey">, GNU<"obfuscation_key">,
↪GNU<"obfuscationkey">,  GNU<"ObfuscationKey">];
+  let Subjects = [Function];
+  let Args = [StringArgument<"Key", 1>];
+}
+
 def ObjCException : InheritableAttr {
   let Spellings = [GNU<"objc_exception">];
 }
--- tools/clang/lib/CodeGen/CodeGenModule.cpp   (revision 192535)
+++ tools/clang/lib/CodeGen/CodeGenModule.cpp   (working copy)
@@ -626,6 +626,10 @@
     B.addAttribute(llvm::Attribute::Cold);
   }
 
+  //HACK: maybe there is a better way to do this
+  if (const ObfKeyAttr *OKA = D->getAttr<ObfKeyAttr>())
+    B.addAttribute("ObfuscationKey",OKA->getKey());
+
   if (D->hasAttr<MinSizeAttr>())
     B.addAttribute(llvm::Attribute::MinSize);
 
--- tools/clang/lib/Sema/SemaDeclAttr.cpp       (revision 192535)
+++ tools/clang/lib/Sema/SemaDeclAttr.cpp       (working copy)
@@ -2759,6 +2759,17 @@
                                                 ParmType,
↪Attr.getLoc()));
 }
 
+static void handleObfKeyAttr(Sema &S, Decl *D, const AttributeList
↪&Attr) {
+  // Make sure that there is a string literal as the sections's
↪single
+  // argument.
+  StringRef Str;
+  SourceLocation LiteralLoc;
+  if (!S.checkStringLiteralArgumentAttr(Attr, 0, Str, &LiteralLoc))
+    return;
+
+  D->addAttr(::new (S.Context) ObfKeyAttr(Attr.getLoc(), S.Context,
```

```
↪Str, Attr.getAttributeSpellingListIndex()));
+}
+
 SectionAttr *Sema::mergeSectionAttr(Decl *D, SourceRange Range,
                                     StringRef Name,
                                     unsigned AttrSpellingListIndex)
↪{
@@ -4646,6 +4657,7 @@
   case AttributeList::AT_InitPriority:
       handleInitPriorityAttr(S, D, Attr); break;

+  case AttributeList::AT_ObfKey:      handleObfKeyAttr      (S, D,
↪Attr); break;
   case AttributeList::AT_Packed:      handlePackedAttr      (S, D,
↪Attr); break;
   case AttributeList::AT_Section:     handleSectionAttr     (S, D,
↪Attr); break;
   case AttributeList::AT_Unavailable:
```

C.3 Obf library

src/Obf/Utils.h

```

#ifndef LLVM_OBF_UTILS_H
#define LLVM_OBF_UTILS_H
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/Support/CommandLine.h"
#include <algorithm>
#include <cstdlib>
#include <cstring>
#include <cstdint>
#include "aes.h"


namespace Obf {
    //Utilities for the Obfuscation transformations
    //These involves mainly things like randomness generators and vector randomization

    //This is the base class of a PRNG, includes some interesting functions
    class PRNG_base {
    protected:
        //Minimal base implementation, generates a string of data
        virtual void get_randoms(char *data, size_t len) = 0;
    public:
        virtual ~PRNG_base() {}
        //Get a random integer
        template <class int_t> int_t get_randomi(int_t end) {
            int_t res;
            get_randoms((char *)&res,sizeof(int_t));
            res %= end;
            //Negative modulos need to be normalized
            res = abs(res);
            return res;
        }
        template <class int_t> bool get_randomb(int_t num, int_t den) {
            int_t rnd = get_randomi(den);
            bool rv = rnd < num;
            return rv;
        }
        template <class int_t> int_t get_randomr(int_t begin, int_t end) {
            return begin + (get_randomi(end-begin));
```

```
        }
        uint64_t rand64() {
            return get_randomi((uint64_t)UINT64_MAX);
        }
        //Randomly rearrange the elements of a vector, uses swaps
↪when available
        template <class RandomAccessIterator> void randomize_vector(
↪RandomAccessIterator first, RandomAccessIterator last) {
            RandomAccessIterator rfirst = first;
            while (last!=first) {
                RandomAccessIterator relem = get_randomr(first,last);
                assert(rfirst <= first && first < last && rfirst <=
↪relem && first <= relem && relem <= last);
                if (first != relem)
                    std::swap(*first, *relem);
                first++;
            }
        }
    };

    //Don't use, it is weak!
    class PRNG_rand : public PRNG_base {
    protected:
        virtual void get_randoms(char *data, size_t len);
    public:
        virtual ~PRNG_rand() {}
    };

    class CPRNG_AES_CTR : public PRNG_base {
        aes_encrypt_ctx cx;
        unsigned char iv[AES_BLOCK_SIZE];
    protected:
        virtual void get_randoms(char *data, size_t len);
    public:
        CPRNG_AES_CTR (const llvm::Function &F, llvm::StringRef gref)
↪;
        CPRNG_AES_CTR (const llvm::Module &M, llvm::StringRef gref);
        static llvm::StringRef get_obf_key(const llvm::Function &F);
        static llvm::StringRef get_obf_key(const llvm::Module &M);
        static void set_obf_key(llvm::Function &F, llvm::StringRef
↪key);
        static void set_obf_key(llvm::Module &M, llvm::StringRef key)
↪;
        static bool has_obf_key(const llvm::Function &F) {
            return !get_obf_key(F).empty();
        }
        static bool has_obf_key(const llvm::Module &M) {
            return !get_obf_key(M).empty();
        }
```

```
            virtual ~CPRNG_AES_CTR() {}
        };

        class Probability {
            uint64_t num;
            uint64_t den;
        public:
            Probability() : num(0), den(1) {
            }
            Probability(uint64_t num, uint64_t den) : num(num), den(den)
↪{
            }
            inline void set(uint64_t num, uint64_t den) {
                this->num = num; this->den = den;
            }
            inline bool roll(PRNG_base &prng) const {
                return prng.get_randomb(num,den);
            }
            inline bool rolldiv(PRNG_base &prng, uint64_t div) const {
                return prng.get_randomb(num,den*div);
            }
        };
        struct ProbabilityParser : public llvm::cl::parser<Probability> {
        // parse - Return true on error.
        bool parse(llvm::cl::Option &O, llvm::StringRef ArgName, const
↪std::string &ArgValue, Probability &Val);
        };

};
#endif
```

src/Obf/Utils.cpp

```
#include <cinttypes>
#include <cstring>
#include "Utils.h"
#include "cmac.h"
#include "llvm/IR/Metadata.h"
#include "llvm/IR/Attributes.h"
#include "llvm/Support/raw_ostream.h"

//Utilities for the Obfuscation transformations
//These involve mainly things like randomness generators and vector
↪randomization
namespace Obf {
    #define emptystringref llvm::StringRef()
    static const char * ObfKeyMDName = "ObfuscationKey";
    static const char * ObfKeyAttrName = "ObfuscationKey";
    static const unsigned char nchar = '\0';
    //Create a key for use with the tag generation algorythm
```

```
    //This is CMAC_cmackey(kid||keydata) where kid is the key type (1 for function 2 for modules) and keydata the keydata

    static void make_cmac_key(unsigned char kid, llvm::StringRef keydata, unsigned char*key) {
        assert(!keydata.empty() && "The obfuscation key shouldn't be empty");
        const static unsigned char cmac_key[16] = {0xab,0xad,0xce,0xba,0xda,0xbe,0xbe,0xfa,0xba,0xda,0xac,0xab,0xac,0xab,0xec,0xea};
        cmac_ctx ctx;
        cmac_init (cmac_key,&ctx);
        cmac_data (&kid,1,&ctx);
        cmac_data ((const unsigned char *)keydata.data(),keydata.size(),&ctx);
        cmac_end (key,&ctx);
    }

    static void make_zero_iv (unsigned char*iv) {
        memset(iv,0,AES_BLOCK_SIZE);
    }

    static void make_tag(unsigned char tid, const unsigned char *key, llvm::StringRef mname, llvm::StringRef fname, llvm::StringRef gref, unsigned char *tag) {
        cmac_ctx ctx;
        //Get the key
        cmac_init ((const unsigned char *)key,&ctx);
        cmac_data (&tid,1,&ctx);
        cmac_data ((const unsigned char *)mname.data(),mname.size(),&ctx);
        cmac_data (&nchar,1,&ctx);
        if (!fname.empty()) {
            cmac_data ((const unsigned char *)fname.data(),fname.size(),&ctx);
            cmac_data (&nchar,1,&ctx);
        }
        assert(!gref.empty() && "The transformation tag shouldn't be empty");
        cmac_data ((const unsigned char *)gref.data(),gref.size(),&ctx);
        cmac_data (&nchar,1,&ctx);
        cmac_end (tag,&ctx);
    }

    llvm::StringRef CPRNG_AES_CTR::get_obf_key(const llvm::Function &F) {
        if(!F.hasFnAttribute(ObfKeyAttrName))
            return emptystringref;
        llvm::Attribute attr =  F.getFnAttribute(ObfKeyAttrName);
```

```
        if (!attr.isStringAttribute())
            return emptystringref;
        return attr.getValueAsString();
    }

    void CPRNG_AES_CTR::set_obf_key(llvm::Function &F, llvm::
↪StringRef key){
        //Replace it
        F.addFnAttr(ObfKeyAttrName,key);
    }

    llvm::StringRef CPRNG_AES_CTR::get_obf_key(const llvm::Module &M)
↪{
        llvm::NamedMDNode *nm = M.getNamedMetadata(ObfKeyMDName);
        if (!nm || nm->getNumOperands() != 1)
            return emptystringref;
        llvm::MDNode *md = nm->getOperand(0);
        if (!md || md->getNumOperands() != 1)
            return emptystringref;
        llvm::MDString * mds = llvm::dyn_cast_or_null<llvm::MDString
↪>(md->getOperand(0));
        if (!mds)
            return emptystringref;
        return mds->getString();
    }

    void CPRNG_AES_CTR::set_obf_key(llvm::Module &M, llvm::StringRef
↪key){
        //First we have to delete the current metadata
        llvm::NamedMDNode *om = M.getNamedMetadata(ObfKeyMDName);
        if (om) {
            M.eraseNamedMetadata(om);
        }
        llvm::NamedMDNode *nm = M.getOrInsertNamedMetadata(
↪ObfKeyMDName);
        assert(nm && "Named␣Metadata␣node␣not␣created");
        assert(nm->getNumOperands() == 0 && "Named␣Metadata␣node␣not␣
↪deleted");
        llvm::MDString * mds = llvm::MDString::get(M.getContext(),key
↪);
        assert(mds && "MDString␣not␣created");
        llvm::MDNode *md = llvm::MDNode::get(M.getContext(),llvm::
↪ArrayRef<llvm::Value *>(mds));
        assert(md && "MDNode␣not␣created");
        nm->addOperand(md);
    }

    //Create a AES_CTR CPRNG object for use in a function
    //The encryption key is created hashing with the CMAC algorithm:
```

```
    //1||ModuleName||0||FunctionName||0||ObfModuleName||0
    CPRNG_AES_CTR::CPRNG_AES_CTR (const llvm::Function &F, llvm::
↪StringRef gref) {
        unsigned char nk[16];
        unsigned char ck[16];
        llvm::StringRef ok = get_obf_key(F);
        assert(!ok.empty() && "No␣obfuscation␣key␣found");
        llvm::StringRef mname = F.getParent()->getModuleIdentifier();
        llvm::StringRef fname = F.getName();
        make_cmac_key(1,ok,ck);
        make_tag(1,ck,mname,fname,gref,nk);
        aes_encrypt_key128(nk,&cx);
        make_zero_iv(iv);
    }

    //Create a AES_CTR CPRNG object for use in a module
    //The encryption key is created hashing with the CMAC algorithm:
    //2||ModuleName||0||ObfModuleName||0
    CPRNG_AES_CTR::CPRNG_AES_CTR (const llvm::Module &M, llvm::
↪StringRef gref) {
        unsigned char nk[16];
        unsigned char ck[16];
        llvm::StringRef ok = get_obf_key(M);
        assert(!ok.empty() && "No␣obfuscation␣key␣found");
        llvm::StringRef mname = M.getModuleIdentifier();
        llvm::StringRef fname = emptystringref;
        make_cmac_key(2,ok,ck);
        make_tag(2,ck,mname,fname,gref,nk);
        aes_encrypt_key128(nk,&cx);
        make_zero_iv(iv);
    }

    void PRNG_rand::get_randoms(char *data, size_t len) {
        for(size_t i=0; i < len ; i++) {
            data[i]=rand();
        }
    }

    //We can generate up to 16 bytes of random data per call, we
↪generate only half of them to make
    //finding the key or the plain text harder in the unlikely case
↪AES is broken
    void CPRNG_AES_CTR::get_randoms(char *data, size_t len) {
        size_t i = 0;
        while( i < len ) {
            unsigned char buf[AES_BLOCK_SIZE];
            AES_RETURN rv;
            rv = aes_ctr_pad(buf, iv, &cx);
            assert(rv == EXIT_SUCCESS && "Failure␣generating␣pseudo␣
```

```
↪random␣data");
            if (len-i < 8)
                memcpy(data+i,buf,len-i);
            else
                memcpy(data+i,buf,8);
            i+=8;
        }
    }

    bool ProbabilityParser::parse(llvm::cl::Option &O, llvm::
↪StringRef ArgName, const std::string &ArgValue, Probability &Val) {
        int nchars;
        uint64_t num, den;
        int rv = sscanf(ArgValue.c_str(),"%" PRIu64 "/%" PRIu64 "%n"
↪,&num,&den,&nchars);
        if (rv != 2 || nchars != (int)ArgValue.size())
            return O.error("'" + ArgValue + "'␣used␣in␣'" + ArgName +
↪ "'␣is␣not␣a␣valid␣probability!");
        Val.set(num,den);
        return false;
    }
};
```

src/Obf/AddModuleKey.cpp

```
#define DEBUG_TYPE "addmodulekey"
#include "llvm/IR/Module.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Pass.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/raw_ostream.h"
#include "Utils.h"
using namespace llvm;
using namespace Obf;

namespace {
    //TODO: generate a random key when none is specified
    static cl::opt< std::string >  TheKey ("modulekey", cl::desc("
↪Specify␣the␣module␣obfuscation␣key"), cl::value_desc("obfkey"), cl
↪::Optional);
    struct AddModuleKey : public ModulePass {
        static char ID; // Pass identification, replacement for
↪typeid
        AddModuleKey() : ModulePass(ID) {}
        virtual bool runOnModule(Module &M){
            if (TheKey.getNumOccurrences() != 1)
                TheKey.error("This␣option␣has␣to␣be␣declared␣when␣
↪using␣the␣addmodulekey␣pass");
            if (TheKey.empty())
                TheKey.error("No␣key␣(or␣an␣empty␣key)␣was␣defined,␣
```

```
↪set␣some␣key");
            CPRNG_AES_CTR::set_obf_key(M, TheKey);
            return true;
        }
    };
}

char AddModuleKey::ID = 0;
static RegisterPass<AddModuleKey> X("addmodulekey", "Add␣the␣desired␣
↪obfuscation␣key␣to␣the␣module␣requires␣the␣-modulekey␣<modulekey>␣
↪option␣set");
```

src/Obf/BBSplit.cpp

```
#define DEBUG_TYPE "bbsplit"
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/InstrTypes.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/User.h"
#include "llvm/Pass.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"
#include "llvm/Analysis/LoopInfo.h"
#include "llvm/Analysis/Dominators.h"
#include "llvm/ADT/Twine.h"
#include "Utils.h"
#include <vector>
using namespace llvm;

STATISTIC(BBSplitCounter, "Number␣of␣basic␣blocks␣splitted");


namespace {
    static Obf::Probability initialProbability(1,16);
    static cl::opt< Obf::Probability, false, Obf::ProbabilityParser >
↪  splitProbability ("splitprobability", cl::desc("Specify␣the␣
↪probability␣of␣splitting␣a␣BB"), cl::value_desc("probability"), cl
↪::Optional, cl::init(initialProbability));
    typedef std::vector<BasicBlock*> blist;
    struct BBSplit : public FunctionPass {
        static char ID; // Pass identification, replacement for
↪typeid
        BBSplit() : FunctionPass(ID) {
        }

        virtual bool runOnFunction(Function &F) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(F))
                return false;
```

```

            Obf::CPRNG_AES_CTR prng(F,"bbsplit");
            bool rval = false;
            blist BBlist;
            BBlist.reserve(F.size());
            //Fill the vector to prevent iterator invalidation
            for (Function::iterator B = F.begin(); B != F.end(); B++)
                BBlist.push_back(B);
            for (blist::iterator B = BBlist.begin(); B != BBlist.end(); B++) {
                BasicBlock * cbb = *B;
                unsigned splitcnt=1;
                //We go to the instruction after the first (if any)
                for (BasicBlock::iterator I=((*B)->getFirstInsertionPt())++; I != cbb->end(); I++) {
                    if (splitProbability.roll(prng)) {
                        cbb=SplitBlock(cbb, I, this);
                        cbb->setName(Twine((*B)->getName(),".rsplit")+Twine(splitcnt));
                        //Again get then next instruction after the one where we did the split
                        I=(cbb->getFirstInsertionPt())++;
                        rval=true;
                        splitcnt++;
                        ++BBSplitCounter;
                    }
                }
            }
            return rval;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.addPreserved<LoopInfo>();
            AU.addPreserved<DominatorTree>();
        }
    };
}

char BBSplit::ID = 0;
static RegisterPass<BBSplit> X("bbsplit", "Randomly split basic blocks in two");
```

src/Obf/FlattenControl.cpp

```
#define DEBUG_TYPE "flattencontrol"
#include "llvm/IR/InstrTypes.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/User.h"
```

```
#include "llvm/Pass.h"
#include "llvm/IR/Constants.h"
#include "llvm/Transforms/Utils/UnifyFunctionExitNodes.h"
#include "llvm/ADT/Twine.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/DenseMap.h"
#include "llvm/Transforms/Utils/BasicBlockUtils.h"
#include "Utils.h"
#include <vector>
#include <cstdint>
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
using namespace Obf;

namespace {
    typedef SmallVector<BasicBlock*,128> bblist;
    typedef std::vector<Use*> uselist;
    typedef SmallPtrSet<BasicBlock*,128> bbset;
    typedef SmallDenseMap<BasicBlock*, ConstantInt*, 128> bb2id;
    struct FlattenControl : public FunctionPass {
        static char ID; // Pass identification, replacement for
↪typeid
        FlattenControl() : FunctionPass(ID) {}
        void generateBBIDs(Function &F, bblist &sbbs, bb2id &bbids,
↪PRNG_base  &prng) const {
            //This function generates an unique identifier for each
↪BB with incoming branches
            //The identifiers follow a set of properties to make the
↪mainblock jump more efficient
            //In particular they go from 0 to the number of blocks-1
↪to which jumps are possible
            // so tables can be compact
            //Works better if called before creating the mainblock
↪itself
            //Step 1 create a set with all the target bbs we can
↪reach
            bbset bbs;
            bblist bbsv;
            for (bblist::iterator B = sbbs.begin(), e = sbbs.end(); B
↪ != e; B++) {
                TerminatorInst *t = (*B)->getTerminator();
                for (unsigned i = 0, e = t->getNumSuccessors(); i !=
↪e; i++) {
                    bbs.insert(t->getSuccessor(i));
                }
            }
            //Convert it into a vector
            bbsv.reserve(bbs.size());
```

```
            for (bbset::iterator i=bbs.begin(),e=bbs.end(); i != e; i
↪++) {
                bbsv.push_back(*i);
            }
            //Randomize it
            prng.randomize_vector(bbsv.begin(),bbsv.end());
            //And finally associate the element position to each
↪block on the bb2id
            int32_t id = 0;
            for (bblist::iterator i=bbsv.begin(),e=bbsv.end(); i != e
↪; i++) {
                bbids[*i]=ConstantInt::get(F.getContext(), APInt(32,
↪id));
                id++;
            }
            return;
        }
        virtual bool runOnFunction(Function &F) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(F))
                return false;

            CPRNG_AES_CTR prng(F,"flattencontrol");
            bblist BranchBlocks;
            //Ensure our entry point contains only the branch
↪instruction
            BasicBlock* newentry = &(F.getEntryBlock());
            {
                BasicBlock* entry = newentry;
                TerminatorInst *t = entry->getTerminator();
                BranchInst *BI = dyn_cast<BranchInst>(t);
                if (&*(entry->getFirstInsertionPt()) != t || !BI ||
↪BI->isConditional()) {
                    newentry = BasicBlock::Create(F.getContext(), "
↪newentry",&F,entry);
                    BranchInst::Create(entry,newentry);
                }
            }
            //The blocks we will process, this ensures iterators don'
↪t break entry is included
            BranchBlocks.reserve(F.size()); //Number of blocks + the
↪new entry block
            for (Function::iterator B = F.begin(); B != F.end(); B++)
                BranchBlocks.push_back(B);
            UnifyFunctionExitNodes &UFEN = getAnalysis<
↪UnifyFunctionExitNodes>();
            //Create the unreachable block for the switch (if one isn
↪'t already there)
            BasicBlock* unr = UFEN.getUnreachableBlock();
```

```
            if (!unr) {
                //If we create this node we don't want it on the list
↪ processed by the algorithm hence the position
                unr = BasicBlock::Create(F.getContext(), "
↪UnifiedUnreachableBlock", &F);
                new UnreachableInst(F.getContext(), unr);
            }
            BasicBlock* main_node = BasicBlock::Create(F.getContext()
↪, "mainblock",&F,++Function::iterator(newentry));
            //Used later, moved here for efficiency
            {  //Keep the live of the list limited
                uselist ul;
                //Check all the uses of the operands, if they are
↪instructions outside of our basic block or the main block or are
↪phis
                //we add a phi for them on the main block so uses
↪dominate users :)
                for(bblist::iterator Bi = BranchBlocks.begin(); Bi !=
↪ BranchBlocks.end(); Bi++) {
                    BasicBlock *B = *Bi;
                    TerminatorInst *TI=B->getTerminator();
                    for(BasicBlock::iterator It = B->begin(); It != B
↪->end(); It++) {
                        //Generate a list of the uses we are
↪interested in
                        //These are non phi uses outside of the BB
↪and the BB terminator
                        bool keepvalues = false;
                        ul.reserve(It->getNumUses()); //Ensure enough
↪ space is available
                        for (Value::use_iterator Ut = It->use_begin()
↪; Ut != It->use_end(); Ut++) {
                            Use *U = &(Ut.getUse());
                            Instruction *I = dyn_cast<Instruction>(U
↪->getUser());
                            if (I == 0) // Not a instruction so we
↪don't care
                                continue;
                            //It is a PHINode, these are handled in a
↪ different way
                            if (isa<PHINode>(*I)) {
                                PHINode * pn = cast<PHINode>(I);
                                //We only want to ignore it if it
↪refers to this block (so the instruction will be used instead)
                                if (pn->getIncomingBlock(*U) == B)
                                    continue;
                                keepvalues = true;
                            } else
                            //If we refer to it from another block we
```

```
↪ need to keep the phi values, for now we do this always but the
↪algorithm can be improved
                            if (I->getParent() != B)
                                keepvalues = true;
                            else if (I != TI || isa<BranchInst>(TI))
↪// Same block and not the terminator, ignore the instruction
                                continue;
                            ul.push_back(U);
                        }
                        //There is at least one interesting use
                        if (!ul.empty()) {
                            Type *type = It->getType();
                            UndefValue *undef = UndefValue::get(type)
↪;
                            //TODO optimize so it won't always keep
↪the variables
                            PHINode *phi = PHINode::Create(type,
↪BranchBlocks.size(), Twine(It->getName(),".uses"), main_node);
                            Value *def = undef; //The default value,
↪if no need to keep it it will be undefined
                            if (keepvalues)
                                def = phi; //Keep the value
                            for (bblist::iterator Bj = BranchBlocks.
↪begin(); Bj != BranchBlocks.end(); Bj++) {
                                BasicBlock *Bjp = *Bj;
                                if (B == Bjp)
                                    phi->addIncoming(&*It, Bjp); //
↪Assign the value
                                else if (Bjp == newentry)
                                    phi->addIncoming(undef, Bjp); //
↪Undefined if it comes from the entry
                                else
                                    phi->addIncoming(def, Bjp); //
↪Keep or undef
                            }
                            //Replace the uses
                            for (uselist::iterator U = ul.begin(); U
↪!= ul.end(); U++) {
                                (*U)->set(phi);
                            }
                            ul.clear();
                        }
                    }
                }
            }
            //Go through our block list moving phis to the core block
↪.
            //Order of the phis doesn't matter as they always refer
↪to the predecessor block variables
```

```
            for(bblist::iterator Bi = BranchBlocks.begin(); Bi != BranchBlocks.end(); Bi++) {
                BasicBlock *B = *Bi;
                PHINode *newphi;
                BasicBlock::iterator Ii=B->begin();
                while (isa<PHINode>(*Ii)) {
                    bbset oldbbs;
                    PHINode *oldphi = cast<PHINode>(Ii);
                    Type *type = oldphi->getType();
                    newphi = PHINode::Create(type, BranchBlocks.size(), "", main_node);
                    //Take the name of the old phi
                    newphi->takeName(oldphi);
                    //Parse and move entries from the phi node (first pass)
                    for (User::op_iterator O = oldphi->op_begin(); O != oldphi->op_end(); O++) {
                        BasicBlock *oldbb;
                        oldbb = oldphi->getIncomingBlock(*O);
                        newphi->addIncoming(*O,oldbb);
                        oldbbs.insert(oldbb);
                    }
                    //Fill it the rest with keeps (second pass)
                    for (bblist::iterator Bj = BranchBlocks.begin(); Bj != BranchBlocks.end(); Bj++) {
                        BasicBlock *Bjp = *Bj;
                        if (oldbbs.count(Bjp))
                            continue;
                        //TODO: we should improve this to make undef if usage is impossible
                        if (Bjp == newentry)
                            newphi->addIncoming(UndefValue::get(type),Bjp); //Undefined if it comes from the entry
                        else
                            newphi->addIncoming(newphi, Bjp); //Keep the value for future references
                    }
                    ReplaceInstWithValue(B->getInstList(), Ii, newphi);
                }
            }
            //Split blocks with terminators we don't know how to handle to get a br we know how to handle
            for(bblist::iterator Bi = BranchBlocks.begin(); Bi != BranchBlocks.end(); Bi++) {
                BasicBlock *B = *Bi;
                TerminatorInst *t = B->getTerminator();
                //Split the non conditional branches
                if (!isa<BranchInst>(*t)) {
```

```
                    //TODO: it is better if we get as much of the flow control as possible here
                    //Use faster function since the transformation breaks the analysis anyways
                    B->splitBasicBlock(t,Twine(B->getName(),".tflat"));
                }
            }
            //Generate the list of possible destinations for our blocks
            bb2id bbids;
            generateBBIDs(F,BranchBlocks,bbids,prng);
            PHINode *phi = PHINode::Create(IntegerType::get(F.getContext(), 32), BranchBlocks.size(), "mainphi", main_node);
            for(bblist::iterator Bi = BranchBlocks.begin(); Bi != BranchBlocks.end(); Bi++) {
                Value *phiv;
                BasicBlock *B = *Bi;
                TerminatorInst *t = B->getTerminator();
                BranchInst *BI = dyn_cast<BranchInst>(t);
                if (BI && BI->isConditional()) {
                    //We associate a number with the destination
                    BasicBlock * s0 = BI->getSuccessor(0), *s1 = BI->getSuccessor(1);
                    assert(bbids.count(s0) && "Successor 0 not on the list");
                    assert(bbids.count(s1) && "Successor 1 not on the list");
                    phiv = SelectInst::Create (BI->getCondition(), bbids[s0], bbids[s1], Twine(B->getName(),".br_select"), t);
                } else {
                    BasicBlock * s = BI->getSuccessor(0);
                    assert(bbids.count(s) &&  "Successor not on the list");
                    //We add the number to the PHI in the main_node
                    phiv=bbids[s];
                }
                //Add the value to the phi
                phi->addIncoming(phiv, B);
                //We make the block branch to the core block
                ReplaceInstWithInst(t, BranchInst::Create(main_node));
            }
            //Now the switch instruction
            SwitchInst *sw = SwitchInst::Create(phi, unr, bbids.size(), main_node);
            for (bb2id::iterator i=bbids.begin(),e=bbids.end();i != e; i++) {
                    sw->addCase(i->second,i->first);
```

```
            }
            return true;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.addRequired<UnifyFunctionExitNodes>(); //Passing this improves the resulting code a lot
        }
    };
}

char FlattenControl::ID = 0;
static RegisterPass<FlattenControl> X("flattencontrol", "Flatten all the nodes to a single node to offuscate the code");
```

src/Obf/ObfuscateConstants.cpp

```
#define DEBUG_TYPE "obfuscateconstants"
#include "llvm/IR/InstrTypes.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/User.h"
#include "llvm/Pass.h"
#include "llvm/ADT/APInt.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Constants.h"
#include <vector>
#include <cstdint>
#include <Utils.h>
#include "llvm/Support/raw_ostream.h"
using namespace llvm;

STATISTIC(ObfuscatedPHIs, "Number of phis with constants obfuscated");
STATISTIC(ObfuscatedIns, "Number of instructions with constants obfuscated");
STATISTIC(ObfuscatedUses, "Number of constants uses obfuscated");
STATISTIC(ObfuscatedCons, "Number of constants obfuscated");
STATISTIC(ReobfuscatedCons, "Number of constants reobfuscated (obfuscated after obfuscation)");

namespace {
    static Obf::Probability initialProbability(1,10);
    static cl::opt< Obf::Probability, false, Obf::ProbabilityParser > reobfuscationProbability ("reobfuscationprobability", cl::desc("Specify the probability of obfuscating again a constant"), cl::value_desc("probability"), cl::Optional, cl::init(initialProbability));
```

```
    //The real pass putting it here makes some code simpler
    class DoObfuscateConstants {
        Module &M;
        GlobalVariable *intC;
        IntegerType *intTy;
        PointerType *intTyPtr;
        std::vector<Constant *> intVs;
        unsigned typelength;
        Obf::CPRNG_AES_CTR *prng;
        inline bool runOnFunction(Function &F) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(F))
                return false;

            bool rval = false;
            prng = new Obf::CPRNG_AES_CTR(F,"obfuscateconstants");
            for (Function::iterator B = F.begin(); B != F.end(); B++)
↪ {
                for (BasicBlock::iterator I=B->begin(); isa<PHINode
↪>(*I); I++)
                    if(runOnPHI(*cast<PHINode>(&*I))) {
                        ObfuscatedPHIs++;
                        rval = true;
                    }
                for (BasicBlock::iterator I=B->getFirstInsertionPt();
↪ I != B->end(); I++)
                    if(runOnNonPHI(*I)) {
                        ObfuscatedIns++;
                        rval = true;
                    }
            }
            delete prng;
            return rval;
        }
        /*obfuscate a constant by introducing instructions before the
↪ insertionPoint*/
        inline Value * obfuscateConstant (Constant &C, Instruction *
↪insertBefore) {
            /*TODO:As of now we can only obfuscate these*/
            if(isa<ConstantInt>(C)) {
                ObfuscatedCons++;
                ConstantInt &IC = cast<ConstantInt>(C);
                if (IC.getType()->getBitWidth() <= typelength && prng
↪->get_randomb(1,2)) {
                    //Obfuscation technique 1: search for the
↪constant in a vector
                    ConstantInt *Cptr = ConstantInt::get(intTy,intVs.
↪size());
                    intVs.push_back(ConstantInt::get(intTy,IC.
```

```
↪getValue().zextOrSelf(typelength)));
                    Value *Vptr = Cptr;
                    if (reobfuscationProbability.roll(*prng)) {
                        ReobfuscatedCons++;
                        Vptr = obfuscateConstant(*Cptr,insertBefore);
                    }
                    LoadInst *lic = new LoadInst(intC, "", false,
↪insertBefore);
                    GetElementPtrInst* ptr = GetElementPtrInst::
↪Create(lic, Vptr,"",insertBefore);
                    LoadInst *li = new LoadInst(ptr, "", false,
↪insertBefore);
                    if (IC.getType()->getBitWidth() == typelength)
                        return li;
                    else return new TruncInst(li, IC.getType(), "",
↪insertBefore);
                } else {
                    //Obfuscation technique 2: replace constant by an
↪ addition or substraction etc of two other constants
                    ConstantInt *C1 = ConstantInt::get(IC.getType(),
↪prng->rand64());
                    //Maybe keep obfuscating the new constant
                    Value *V1 = C1;
                    if (reobfuscationProbability.rolldiv(*prng,2)) {
                        ReobfuscatedCons++;
                        V1 = obfuscateConstant(*C1,insertBefore);
                    }
                    APInt VC2;
                    Instruction::BinaryOps op;
                    //Basic example, we only use Add sub or xor since
↪ muls ands and ors are more complicated
                    switch (prng->get_randomi(3)) {
                        case 0:
                            VC2 = IC.getValue()-C1->getValue();
                            op=Instruction::Add;
                            assert((VC2 + C1->getValue())==IC.
↪getValue());
                            break;
                        case 1:
                            VC2 = IC.getValue()+C1->getValue();
                            op=Instruction::Sub;
                            assert((VC2 - C1->getValue())==IC.
↪getValue());
                            break;
                        case 2:
                            VC2 = IC.getValue()^C1->getValue();
                            op=Instruction::Xor;
                            assert((VC2 ^ C1->getValue())==IC.
↪getValue());
```

```
                                break;
                            }
                            ConstantInt *C2 = cast<ConstantInt>(ConstantInt::
↪get(IC.getType(),VC2));
                            Value *V2 = C2;
                            //Maybe keep obfuscating the new constant
                            if (reobfuscationProbability.rolldiv(*prng,2)) {
                                ReobfuscatedCons++;
                                V2 = obfuscateConstant(*C2,insertBefore);
                            }
                            return BinaryOperator::Create(op, V2, V1, "",
↪insertBefore);
                        }
                        //TODO: obfuscation technique 3: use a formula
↪returning the constant over some previous value
                    }
                    return &C;
                }
                //Obfuscate an Use if it s a constant (and we want to do so)
                //Returns true if the use was modified
                inline bool obfuscateUse(Use &U, Instruction *insertBefore) {
                    Constant *C = dyn_cast<Constant>(U.get());
                    if (C == 0) return false;//Not a constant
                    Value *NC = obfuscateConstant(*C, insertBefore);
                    if (NC == C) return false; //The constant wasn't modified
                    ObfuscatedUses++;
                    U.set(NC);
                    return true;
                }
                /* Run on a phi instruction */
                inline bool runOnPHI(PHINode &phi) {
                    bool rval = false;
                    /*If a constant is found the value must be calculated on
↪the phy node bringing us here*/
                    for (User::op_iterator O = phi.op_begin(); O != phi.
↪op_end(); O++) {
                        rval |= obfuscateUse(*O,phi.getIncomingBlock(*O)->
↪getTerminator());
                    }
                    return rval;
                }
                /* Run on a non phi instruction*/
                inline bool runOnNonPHI(Instruction &I) {
                    bool rval=false;
                    /*Check only value (arg 1)*/
                    if(isa<SwitchInst>(I))
                        return obfuscateUse(I.getOperandUse(0),&I);
                    /*Check only vectors (args 1 and 2)*/
                    if(isa<ShuffleVectorInst>(I))
```

```
                    return obfuscateUse(I.getOperandUse(0),&I) |
↪obfuscateUse(I.getOperandUse(1),&I);
                /*Check only struct and value (args 1 and 2)*/
                if(isa<InsertValueInst>(I))
                    return obfuscateUse(I.getOperandUse(0),&I) |
↪obfuscateUse(I.getOperandUse(1),&I);
                /*Check only struct (arg 1)*/
                if(isa<ExtractValueInst>(I))
                    return obfuscateUse(I.getOperandUse(0),&I);
                /*Check only NumElements (arg 1)*/
                if(isa<AllocaInst>(I))
                    return obfuscateUse(I.getOperandUse(0),&I);
                /*Ignore alignment*/
                if(isa<LoadInst>(I))
                    return obfuscateUse(I.getOperandUse(0),&I);
                /*TODO: Ignore constants in structs*/
                if(isa<GetElementPtrInst>(I))
                    return false;
                /*landingpads???*/
                /*Intrinsics some lifetime ie give problems*/
                if(isa<CallInst>(I))
                    return false;
                /*Check all the values*/
                for (User::op_iterator O = I.op_begin(); O != I.op_end();
↪ O++) {
                    rval |= obfuscateUse(*O,&I);
                }
                return rval;
            }
        public:
            DoObfuscateConstants(Module &M) : M(M) {
            }
            bool run() {
                //TODO:This should depend on the target type
                typelength=64;
                intTy = IntegerType::get(M.getContext(), typelength);
                intTyPtr = PointerType::get(intTy, 0);

                intC = new GlobalVariable(M,intTyPtr,false,GlobalVariable
↪::PrivateLinkage,0,".data");
                bool rval = false;
                for (Module::iterator F = M.begin(); F != M.end(); F++) {
                    if (F->empty())
                        continue;
                    rval |= runOnFunction(*F);
                }
                //TODO: check deeply the possibility of getting rid of
↪the pointer memory access by using a placeholder
                ArrayType* ArrayTy = ArrayType::get(intTy, intVs.size());
```

```
                GlobalVariable *arrC = new GlobalVariable(M,ArrayTy,false,GlobalVariable::PrivateLinkage,ConstantArray::get(ArrayTy, intVs));
                intC->setInitializer(ConstantExpr::getGetElementPtr(arrC, std::vector<Constant*>(2,ConstantInt::get(intTy,0))));
                return rval;
            }
    };
    //Saddly we can't keep global state if using the FunctionPass :(
    struct ObfuscateConstants : public ModulePass {
        static char ID; // Pass identification, replacement for typeid
        ObfuscateConstants() : ModulePass(ID) {}
        virtual bool runOnModule(Module &M){
            DoObfuscateConstants obc(M);
            return obc.run();
        }
    };
}

char ObfuscateConstants::ID = 0;
static RegisterPass<ObfuscateConstants> X("obfuscateconstants", "Obfuscate the code constants by converting them into mathematical operations and dereferences from a vector");
```

src/Obf/PropagateModuleKey.cpp

```
#define DEBUG_TYPE "propagatemodulekey"
#include "llvm/IR/Module.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Pass.h"
#include "llvm/Support/raw_ostream.h"
#include "Utils.h"
using namespace llvm;
using namespace Obf;

namespace {
    //TODO: add flag to decide whether overwrite keys, merge them or keep them
    //For now we just replace
    struct PropagateModuleKey : public FunctionPass {
        static char ID; // Pass identification, replacement for typeid
        PropagateModuleKey() : FunctionPass(ID) {}
        virtual bool runOnFunction(Function &F) {
            //if no module key found just leave it alone
            if (!CPRNG_AES_CTR::has_obf_key(*(F.getParent())))
                return false;
            StringRef TheKey = CPRNG_AES_CTR::get_obf_key(*(F.
```

```
↪getParent()));
            CPRNG_AES_CTR::set_obf_key(F, TheKey);
            return true;
        }
    };
}

char PropagateModuleKey::ID = 0;
static RegisterPass<PropagateModuleKey> X("propagatemodulekey", "
↪Propagate␣the␣module␣obfuscation␣key␣to␣the␣function");
```

src/Obf/RandBB.cpp

```
#define DEBUG_TYPE "randbb"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/User.h"
#include "llvm/Pass.h"
#include "llvm/ADT/SmallVector.h"
#include <algorithm>
#include "Utils.h"
using namespace llvm;

namespace {
    typedef SmallVector<BasicBlock*,128> candmap;
    struct RandBB : public FunctionPass {
        static char ID; // Pass identification, replacement for
↪typeid
        RandBB() : FunctionPass(ID) {}

        virtual bool runOnFunction(Function &F) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(F))
                return false;

            Obf::CPRNG_AES_CTR prng(F,"randbb");
            candmap candidates; //List of candidates
            //Exclude the entry block
            Function::iterator src=F.getEntryBlock();
            candidates.reserve(F.size()-1);
            //First pass, initialize structures
            for (Function::iterator B = F.begin(), e = F.end(); B !=
↪e ; B++) {
                if (B != src) { //DO NOT MOVE THE ENTRY POINT!
                    candidates.push_back(B);
                }
            }
            prng.randomize_vector(candidates.begin(),candidates.end()
↪);
            for (candmap::size_type i = 0; i < candidates.size(); i
```

```
↪++) {
                //Pick an element from the list
                BasicBlock *dst=candidates[i];

                if (dst != src) { //If swap is needed
                    dst->moveAfter(src);
                    src = dst;
                }
            }
            return true;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };
}

char RandBB::ID = 0;
static RegisterPass<RandBB> X("randbb", "Randomly␣rearrange␣BBs␣
↪inside␣functions␣keeping␣the␣entry␣point");
```

src/Obf/RandFun.cpp

```
#define DEBUG_TYPE "randfun"
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/Pass.h"
#include "llvm/ADT/SmallVector.h"
#include <algorithm>
#include "llvm/Support/raw_ostream.h"
#include "Utils.h"
using namespace llvm;

namespace {
    typedef SmallVector<Function*,128> candmap;
    struct RandFun : public ModulePass {
        static char ID; // Pass identification, replacement for
↪typeid
        RandFun() : ModulePass(ID) {}

        virtual bool runOnModule(Module &M) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(M))
                return false;

            Obf::CPRNG_AES_CTR prng(M,"randfun");
            candmap candidates; //List of candidates
            Module::iterator src=M.begin();
            Module::FunctionListType &fl=M.getFunctionList();
            candidates.reserve(M.size());
```

```
            //First pass, initialize structures
            for (Module::iterator F = src, e = M.end(); F != e ; F++)
↪ {
                candidates.push_back(F);
            }
            prng.randomize_vector(candidates.begin(),candidates.end()
↪);
            for (candmap::size_type i = 0; i < candidates.size(); i
↪++) {
                //Pick an element from the list
                Function *dst=candidates[i];

                if (dst != src) { //If swap is needed
                    fl.remove(dst);
                    fl.insert(src,dst);
                } else {
                    src++;
                }
            }
            return true;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };
}

char RandFun::ID = 0;
static RegisterPass<RandFun> X("randfun", "Randomly␣rearrange␣
↪functions␣inside␣a␣module");
```

src/Obf/RandGlb.cpp

```
#define DEBUG_TYPE "randglb"
#include "llvm/IR/GlobalVariable.h"
#include "llvm/IR/Module.h"
#include "llvm/Pass.h"
#include "llvm/ADT/SmallVector.h"
#include <algorithm>
#include "llvm/Support/raw_ostream.h"
#include "Utils.h"
using namespace llvm;

namespace {
    typedef SmallVector<GlobalVariable*,128> candmap;
    struct RandGlb : public ModulePass {
        static char ID; // Pass identification, replacement for
↪typeid
        RandGlb() : ModulePass(ID) {}

```

```cpp
        virtual bool runOnModule(Module &M) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(M))
                return false;

            Obf::CPRNG_AES_CTR prng(M,"randglb");
            candmap candidates; //List of candidates
            Module::global_iterator src=M.global_begin();
            Module::GlobalListType &gl=M.getGlobalList();
            candidates.reserve(gl.size());
            //First pass, initialize structures
            for (Module::global_iterator G = src, e = M.global_end();
↪ G != e ; G++) {
                candidates.push_back(G);
            }
            prng.randomize_vector(candidates.begin(),candidates.end()
↪);
            for (candmap::size_type i = 0; i < candidates.size(); i
↪++) {
                //Pick an element from the list
                GlobalVariable *dst=candidates[i];

                if (dst != src) { //If swap is needed
                    gl.remove(dst);
                    gl.insert(src,dst);
                } else {
                    src++;
                }
            }
            return true;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };
}

char RandGlb::ID = 0;
static RegisterPass<RandGlb> X("randglb", "Randomly␣rearrange␣global␣
↪variables␣inside␣a␣module");
```

src/Obf/RandIns.cpp

```cpp
#define DEBUG_TYPE "randins"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/User.h"
#include "llvm/Pass.h"
```

```
#include "llvm/ADT/DenseMap.h"
#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/SmallPtrSet.h"
#include <algorithm>
#include "Utils.h"
using namespace llvm;

namespace {
    typedef SmallPtrSet<Instruction*,16> deplst;
    typedef SmallDenseMap<Instruction*,deplst,256> depmap;
    typedef SmallVector<Instruction*,256> candmap;
    struct RandIns : public FunctionPass {
        static char ID; // Pass identification, replacement for typeid
        RandIns() : FunctionPass(ID) {}

        virtual bool runOnFunction(Function &F) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(F))
                return false;

            Obf::CPRNG_AES_CTR prng(F,"randins");
            candmap candidates; //List of candidates
            depmap dependencies;
            unsigned dsti;
            BasicBlock::iterator src;
            for (Function::iterator B = F.begin(), e = F.end(); B != e ; B++) {
                //First pass, initialize structures
                //Map each element to its position on the list (and the other way around)
                candidates.reserve(B->size());
                src=B->begin();
                for (BasicBlock::iterator I=src, e = BasicBlock::iterator(B->getFirstNonPHI()); I != e; I++) {
                    candidates.push_back(I);
                }
                prng.randomize_vector(candidates.begin(),candidates.end());
                //Reorder Phi Nodes randomly
                for (candmap::size_type i = 0; i < candidates.size(); i++) {
                    Instruction *dst=candidates[i];
                    if (dst != src) { //If swap is needed
                        dst->moveBefore(src);
                    } else {
                        src++;
                    }
                }
```

```
                candidates.clear();
                candidates.reserve(B->size());
                dependencies.grow(B->size());
                //Generate the dependency map
                src= B->getFirstInsertionPt();
                for (BasicBlock::iterator I=src, e  = BasicBlock::iterator(B->getTerminator()); I != e; I++) {
                    //Check the dependency map
                    deplst *lst = NULL;
                    //If the instruction may have undesired side effects make it block other stuff with side effects or memory accesses
                    if (I->mayHaveSideEffects()) for (BasicBlock::iterator J=I; ++J != e;) {
                        if (J->mayReadFromMemory() || J->mayHaveSideEffects())
                            dependencies[J].insert(I);
                    }
                    if (I->mayReadFromMemory()) for (BasicBlock::iterator J=I; ++J != e;) {
                        if (J->mayHaveSideEffects())
                            dependencies[J].insert(I);
                    }
                    for (User::op_iterator U = I->op_begin(), e= I->op_end(); U != e; ++U) {
                        Instruction *i = dyn_cast<Instruction>(*U);
                        if (!i)
                            continue;
                        if (isa<PHINode>(*i))
                            continue;
                        if (i->getParent() != B)
                            continue;
                        //Get the deplst
                        if (!lst)
                            lst = &(dependencies[I]);
                        //Insert use on the map
                        lst->insert(i);
                    }
                    if (!lst) {
                        candidates.push_back(I);
                    }
                }
                //TODO: this should be done by the prng class given the dependency list
                //Reorder Instructions randomly
                while (!candidates.empty()) {
                    //Pick a random element from the list
                    dsti = prng.get_randomi(candidates.size());
                    Instruction *dst=candidates[dsti];
```

```
                    if (dst != src) { //If swap is needed
                        dst->moveBefore(src);
                    } else {
                        src++;
                    }
                    candidates[dsti]=candidates.back();
                    candidates.pop_back();
                    //Remove the use from the dependencies
                    for (Value::use_iterator It = dst->use_begin(), e
↪ = dst->use_end(); It != e; ++It) {
                        Instruction *i = dyn_cast<Instruction>(*It);
                        depmap::iterator p = dependencies.find(i);
                        if ( p != dependencies.end() ) {
                            p->second.erase(dst);
                            if (p->second.empty()) {
                                dependencies.erase(p);
                                candidates.push_back(i);
                            }
                        }
                    }
                }
                dependencies.shrink_and_clear();
            }
            return true;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };
}

char RandIns::ID = 0;
static RegisterPass<RandIns> X("randins", "Randomly␣rearrange␣
↪instructions␣inside␣BBs␣keeping␣dependences");
```

src/Obf/SwapOps.cpp

```
#define DEBUG_TYPE "swapops"
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/InstrTypes.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/User.h"
#include "llvm/Pass.h"
#include "Utils.h"
using namespace llvm;

STATISTIC(SwapCounter, "Number␣of␣operands␣swapped");

```

```
namespace {
    struct SwapOps : public FunctionPass {
        static char ID; // Pass identification, replacement for typeid
        SwapOps() : FunctionPass(ID) {}

        virtual bool runOnFunction(Function &F) {
            //if no module key found just leave the function alone
            if (!Obf::CPRNG_AES_CTR::has_obf_key(F))
                return false;

            bool rval = false;
            Obf::CPRNG_AES_CTR prng(F,"swapops");
            for (Function::iterator B = F.begin(); B != F.end(); B++) {
                for (BasicBlock::iterator I=B->getFirstInsertionPt(); I != B->end(); I++) {
                    if (!I->isCommutative())
                        continue;
                    BinaryOperator *BO = dyn_cast<BinaryOperator>(I);
                    if (!BO)
                        continue;
                    if (prng.get_randomb(1,2)) {
                        BO->swapOperands();
                        rval=true;
                        ++SwapCounter;
                    }
                }
            }
            return rval;
        }
        void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.setPreservesCFG();
        }
    };
}

char SwapOps::ID = 0;
static RegisterPass<SwapOps> X("swapops", "Randomly swap the operators of commutative binary instructions");
```

C.4 AES library

src/Obf/aes.h

```
/*
---------------------------------------------------------------------------
↪
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights
↪reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied
↪warranties
in respect of its operation, including, but not limited to,
↪correctness
and fitness for purpose.
---------------------------------------------------------------------------
↪
Issue Date: 20/12/2007

 This file contains the definitions required to use AES in C. See
↪aesopt.h
 for optimisation details.
*/

#ifndef _AES_H
#define _AES_H

#include <stdlib.h>

/*  This include is used to find 8 & 32 bit unsigned integer types
↪*/
#include "brg_types.h"

#if defined(__cplusplus)
extern "C"
{
#endif

#define AES_128
#define FIXED_TABLES
```

```c

/* The following must also be set in assembler files if being used
↪*/

#define AES_ENCRYPT /* if support for encryption is needed
↪*/

#define AES_BLOCK_SIZE  16  /* the AES block size in bytes
↪*/
#define N_COLS           4  /* the number of columns in the state
↪*/

/* The key schedule length is 11, 13 or 15 16-byte blocks for 128,
↪*/
/* 192 or 256-bit keys respectively. That is 176, 208 or 240 bytes
↪*/
/* or 44, 52 or 60 32-bit words.
↪*/

#define KS_LENGTH       44

#define AES_RETURN INT_RETURN

/* the character array 'inf' in the following structures is used
↪*/
/* to hold AES context information. This AES code uses cx->inf.b[0]
↪*/
/* to hold the number of rounds multiplied by 16. The other three
↪*/
/* elements can be used by code that implements additional modes
↪*/

typedef union
{   uint_32t l;
    uint_8t b[4];
} aes_inf;

typedef struct
{   uint_32t ks[KS_LENGTH];
    aes_inf inf;
} aes_encrypt_ctx;

/* This routine must be called before first use if non-static
↪*/
/* tables are being used
↪*/

AES_RETURN aes_init(void);

```

```
/* Key lengths in the range 16 <= key_len <= 32 are given in bytes, */
/* those in the range 128 <= key_len <= 256 are given in bits */

AES_RETURN aes_encrypt_key128(const unsigned char *key, aes_encrypt_ctx cx[1]);
AES_RETURN aes_encrypt(const unsigned char *in, unsigned char *out, const aes_encrypt_ctx cx[1]);

/* Multiple calls to the following subroutines for multiple block */
/* ECB, CBC, CFB, OFB and CTR mode encryption can be used to handle */
/* long messages incremantally provided that the context AND the iv */
/* are preserved between all such calls.  For the ECB and CBC modes */
/* each individual call within a series of incremental calls must */
/* process only full blocks (i.e. len must be a multiple of 16) but */
/* the CFB, OFB and CTR mode calls can handle multiple incremental */
/* calls of any length. Each mode is reset when a new AES key is */
/* set but ECB and CBC operations can be reset without setting a */
/* new key by setting a new IV value.  To reset CFB, OFB and CTR */
/* without setting the key, aes_mode_reset() must be called and the */
/* IV must be set.  NOTE: All these calls update the IV on exit so */
/* this has to be reset if a new operation with the same IV as the */
/* previous one is required (or decryption follows encryption with */
/* the same IV array). */

AES_RETURN aes_ecb_encrypt(const unsigned char *ibuf, unsigned char *obuf, int len, const aes_encrypt_ctx ctx[1]);
AES_RETURN aes_ctr_pad(unsigned char *obuf, unsigned char *cbuf, aes_encrypt_ctx cx[1]);

#if defined(__cplusplus)
}
#endif
```

```

#endif
```

src/Obf/aes_modes.c

```
/*
---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 20/12/2007

 These subroutines implement multiple block AES modes for ECB, CBC, CFB,
 OFB and CTR encryption,  The code provides support for the VIA Advanced
 Cryptography Engine (ACE).

 NOTE: In the following subroutines, the AES contexts (ctx) must be
 16 byte aligned if VIA ACE is being used
*/

#include <string.h>
#include <assert.h>
#include <stdio.h>

#include "aesopt.h"

#if defined(__cplusplus)
extern "C"
{
#endif

```

```
#if defined( _MSC_VER ) && ( _MSC_VER > 800 )
#pragma intrinsic(memcpy)
#endif

#define BFR_BLOCKS      8

/* These values are used to detect long word alignment in order to */
/* speed up some buffer operations. This facility may not work on  */
/* some machines so this define can be commented out if necessary  */

#define FAST_BUFFER_OPERATIONS

#define lp32(x)         ((uint_32t*)(x))

#if defined( USE_VIA_ACE_IF_PRESENT )

#include "aes_via_ace.h"

#pragma pack(16)

aligned_array(unsigned long,    enc_gen_table, 12, 16) =
↪NEH_ENC_GEN_DATA;
aligned_array(unsigned long,   enc_load_table, 12, 16) =
↪NEH_ENC_LOAD_DATA;
aligned_array(unsigned long, enc_hybrid_table, 12, 16) =
↪NEH_ENC_HYBRID_DATA;
aligned_array(unsigned long,    dec_gen_table, 12, 16) =
↪NEH_DEC_GEN_DATA;
aligned_array(unsigned long,   dec_load_table, 12, 16) =
↪NEH_DEC_LOAD_DATA;
aligned_array(unsigned long, dec_hybrid_table, 12, 16) =
↪NEH_DEC_HYBRID_DATA;

/* NOTE: These control word macros must only be used after  */
/* a key has been set up because they depend on key size    */
/* See the VIA ACE documentation for key type information   */
/* and aes_via_ace.h for non-default NEH_KEY_TYPE values    */

#ifndef NEH_KEY_TYPE
#  define NEH_KEY_TYPE NEH_HYBRID
#endif

#if NEH_KEY_TYPE == NEH_LOAD
#define kd_adr(c)   ((uint_8t*)(c)->ks)
#elif NEH_KEY_TYPE == NEH_GENERATE
#define kd_adr(c)   ((uint_8t*)(c)->ks + (c)->inf.b[0])
#elif NEH_KEY_TYPE == NEH_HYBRID
#define kd_adr(c)   ((uint_8t*)(c)->ks + ((c)->inf.b[0] == 160 ? 160
↪: 0))
```

```
#else
#error no key type defined for VIA ACE
#endif

#else

#define aligned_array(type, name, no, stride) type name[no]
#define aligned_auto(type, name, no, stride)  type name[no]

#endif

#if defined( _MSC_VER ) && _MSC_VER > 1200

#define via_cwd(cwd, ty, dir, len) \
    unsigned long* cwd = (dir##_##ty##_table + ((len - 128) >> 4))

#else

#define via_cwd(cwd, ty, dir, len)                  \
    aligned_auto(unsigned long, cwd, 4, 16);        \
    cwd[1] = cwd[2] = cwd[3] = 0;                   \
    cwd[0] = neh_##dir##_##ty##_key(len)

#endif

AES_RETURN aes_ecb_encrypt(const unsigned char *ibuf, unsigned char *
↪obuf,
                    int len, const aes_encrypt_ctx ctx[1])
{   int nb = len >> 4;

    if(len & (AES_BLOCK_SIZE - 1))
        return EXIT_FAILURE;

#if defined( USE_VIA_ACE_IF_PRESENT )

    if(ctx->inf.b[1] == 0xff)
    {   uint_8t *ksp = (uint_8t*)(ctx->ks);
        via_cwd(cwd, hybrid, enc, 2 * ctx->inf.b[0] - 192);

        if(ALIGN_OFFSET( ctx, 16 ))
            return EXIT_FAILURE;

        if(!ALIGN_OFFSET( ibuf, 16 ) && !ALIGN_OFFSET( obuf, 16 ))
        {
            via_ecb_op5(ksp, cwd, ibuf, obuf, nb);
        }
        else
        {   aligned_auto(uint_8t, buf, BFR_BLOCKS * AES_BLOCK_SIZE,
↪16);
```

```
            uint_8t *ip, *op;

            while(nb)
            {
                int m = (nb > BFR_BLOCKS ? BFR_BLOCKS : nb);

                ip = (ALIGN_OFFSET( ibuf, 16 ) ? buf : ibuf);
                op = (ALIGN_OFFSET( obuf, 16 ) ? buf : obuf);

                if(ip != ibuf)
                    memcpy(buf, ibuf, m * AES_BLOCK_SIZE);

                via_ecb_op5(ksp, cwd, ip, op, m);

                if(op != obuf)
                    memcpy(obuf, buf, m * AES_BLOCK_SIZE);

                ibuf += m * AES_BLOCK_SIZE;
                obuf += m * AES_BLOCK_SIZE;
                nb -= m;
            }
        }

        return EXIT_SUCCESS;
    }

#endif

#if !defined( ASSUME_VIA_ACE_PRESENT )
    while(nb--)
    {
        if(aes_encrypt(ibuf, obuf, ctx) != EXIT_SUCCESS)
            return EXIT_FAILURE;
        ibuf += AES_BLOCK_SIZE;
        obuf += AES_BLOCK_SIZE;
    }
#endif
    return EXIT_SUCCESS;
}

AES_RETURN aes_ctr_pad(unsigned char *obuf, unsigned char *cbuf,
↪aes_encrypt_ctx ctx[1])
{
#if defined( USE_VIA_ACE_IF_PRESENT )
    if(ctx->inf.b[1] == 0xff && ALIGN_OFFSET( ctx, 16 ))
        return EXIT_FAILURE;
#endif
    int i;
    unsigned char acc;
```

```

    memcpy(obuf, cbuf, AES_BLOCK_SIZE);
    for (acc=1,i = AES_BLOCK_SIZE-1; i >= 0; i--) {
        cbuf[i] += acc;
        acc &= cbuf[i] == 0;
    }
    return aes_ecb_encrypt(obuf, obuf, AES_BLOCK_SIZE, ctx);
}

#if defined(__cplusplus)
}
#endif
```

src/Obf/aes_via_ace.h

```
/*
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 20/12/2007
*/

#ifndef AES_VIA_ACE_H
#define AES_VIA_ACE_H

#if defined( _MSC_VER )
#  define INLINE  __inline
#elif defined( __GNUC__ )
#  define INLINE  static inline
#else
#  error VIA ACE requires Microsoft or GNU C
#endif

#define NEH_GENERATE    1
```

```
#define NEH_LOAD         2
#define NEH_HYBRID       3

#define MAX_READ_ATTEMPTS   1000

/* VIA Nehemiah RNG and ACE Feature Mask Values */

#define NEH_CPU_IS_VIA      0x00000001
#define NEH_CPU_READ        0x00000010
#define NEH_CPU_MASK        0x00000011

#define NEH_RNG_PRESENT     0x00000004
#define NEH_RNG_ENABLED     0x00000008
#define NEH_ACE_PRESENT     0x00000040
#define NEH_ACE_ENABLED     0x00000080
#define NEH_RNG_FLAGS       (NEH_RNG_PRESENT | NEH_RNG_ENABLED)
#define NEH_ACE_FLAGS       (NEH_ACE_PRESENT | NEH_ACE_ENABLED)
#define NEH_FLAGS_MASK      (NEH_RNG_FLAGS | NEH_ACE_FLAGS)

/* VIA Nehemiah Advanced Cryptography Engine (ACE) Control Word
↪Values  */

#define NEH_GEN_KEY     0x00000000      /* generate key schedule
↪         */
#define NEH_LOAD_KEY    0x00000080      /* load schedule from memory
↪   */
#define NEH_ENCRYPT     0x00000000      /* encryption
↪                      */
#define NEH_DECRYPT     0x00000200      /* decryption
↪                      */
#define NEH_KEY128      0x00000000+0x0a /* 128 bit key
↪                     */
#define NEH_KEY192      0x00000400+0x0c /* 192 bit key
↪                     */
#define NEH_KEY256      0x00000800+0x0e /* 256 bit key
↪                     */

#define NEH_ENC_GEN     (NEH_ENCRYPT | NEH_GEN_KEY)
#define NEH_DEC_GEN     (NEH_DECRYPT | NEH_GEN_KEY)
#define NEH_ENC_LOAD    (NEH_ENCRYPT | NEH_LOAD_KEY)
#define NEH_DEC_LOAD    (NEH_DECRYPT | NEH_LOAD_KEY)

#define NEH_ENC_GEN_DATA {\
    NEH_ENC_GEN | NEH_KEY128, 0, 0, 0,\
    NEH_ENC_GEN | NEH_KEY192, 0, 0, 0,\
    NEH_ENC_GEN | NEH_KEY256, 0, 0, 0 }

#define NEH_ENC_LOAD_DATA {\
    NEH_ENC_LOAD | NEH_KEY128, 0, 0, 0,\
```

```
    NEH_ENC_LOAD | NEH_KEY192, 0, 0, 0,\
    NEH_ENC_LOAD | NEH_KEY256, 0, 0, 0 }

#define NEH_ENC_HYBRID_DATA {\
    NEH_ENC_GEN  | NEH_KEY128, 0, 0, 0,\
    NEH_ENC_LOAD | NEH_KEY192, 0, 0, 0,\
    NEH_ENC_LOAD | NEH_KEY256, 0, 0, 0 }

#define NEH_DEC_GEN_DATA {\
    NEH_DEC_GEN | NEH_KEY128, 0, 0, 0,\
    NEH_DEC_GEN | NEH_KEY192, 0, 0, 0,\
    NEH_DEC_GEN | NEH_KEY256, 0, 0, 0 }

#define NEH_DEC_LOAD_DATA {\
    NEH_DEC_LOAD | NEH_KEY128, 0, 0, 0,\
    NEH_DEC_LOAD | NEH_KEY192, 0, 0, 0,\
    NEH_DEC_LOAD | NEH_KEY256, 0, 0, 0 }

#define NEH_DEC_HYBRID_DATA {\
    NEH_DEC_GEN  | NEH_KEY128, 0, 0, 0,\
    NEH_DEC_LOAD | NEH_KEY192, 0, 0, 0,\
    NEH_DEC_LOAD | NEH_KEY256, 0, 0, 0 }

#define neh_enc_gen_key(x)  ((x) == 128 ? (NEH_ENC_GEN | NEH_KEY128) :       \
     (x) == 192 ? (NEH_ENC_GEN | NEH_KEY192) : (NEH_ENC_GEN | NEH_KEY256))

#define neh_enc_load_key(x) ((x) == 128 ? (NEH_ENC_LOAD | NEH_KEY128) :      \
     (x) == 192 ? (NEH_ENC_LOAD | NEH_KEY192) : (NEH_ENC_LOAD | NEH_KEY256))

#define neh_enc_hybrid_key(x)   ((x) == 128 ? (NEH_ENC_GEN | NEH_KEY128) :  \
     (x) == 192 ? (NEH_ENC_LOAD | NEH_KEY192) : (NEH_ENC_LOAD | NEH_KEY256))

#define neh_dec_gen_key(x)  ((x) == 128 ? (NEH_DEC_GEN | NEH_KEY128) :       \
     (x) == 192 ? (NEH_DEC_GEN | NEH_KEY192) : (NEH_DEC_GEN | NEH_KEY256))

#define neh_dec_load_key(x) ((x) == 128 ? (NEH_DEC_LOAD | NEH_KEY128) :      \
     (x) == 192 ? (NEH_DEC_LOAD | NEH_KEY192) : (NEH_DEC_LOAD | NEH_KEY256))

#define neh_dec_hybrid_key(x)   ((x) == 128 ? (NEH_DEC_GEN |
```

```
↪NEH_KEY128) :  \
        (x) == 192 ? (NEH_DEC_LOAD | NEH_KEY192) : (NEH_DEC_LOAD |
↪NEH_KEY256))

#if defined( _MSC_VER ) && ( _MSC_VER > 1200 )
#define aligned_auto(type, name, no, stride)  __declspec(align(stride
↪)) type name[no]
#else
#define aligned_auto(type, name, no, stride)                \
    unsigned char _##name[no * sizeof(type) + stride];      \
    type *name = (type*)(16 * ((((unsigned long)(_##name)) + stride -
↪ 1) / stride))
#endif

#if defined( _MSC_VER ) && ( _MSC_VER > 1200 )
#define aligned_array(type, name, no, stride) __declspec(align(stride
↪)) type name[no]
#elif defined( __GNUC__ )
#define aligned_array(type, name, no, stride) type name[no]
↪__attribute__ ((aligned(stride)))
#else
#define aligned_array(type, name, no, stride) type name[no]
#endif

/* VIA ACE codeword     */

static unsigned char via_flags = 0;

#if defined ( _MSC_VER ) && ( _MSC_VER > 800 )

#define NEH_REKEY   __asm pushfd __asm popfd
#define NEH_AES     __asm _emit 0xf3 __asm _emit 0x0f __asm _emit 0
↪xa7
#define NEH_ECB     NEH_AES __asm _emit 0xc8
#define NEH_CBC     NEH_AES __asm _emit 0xd0
#define NEH_CFB     NEH_AES __asm _emit 0xe0
#define NEH_OFB     NEH_AES __asm _emit 0xe8
#define NEH_RNG     __asm _emit 0x0f __asm _emit 0xa7 __asm _emit 0
↪xc0

INLINE int has_cpuid(void)
{   char ret_value;
    __asm
    {   pushfd                      /* save EFLAGS register     */
        mov     eax,[esp]           /* copy it to eax           */
        mov     edx,0x00200000      /* CPUID bit position       */
        xor     eax,edx             /* toggle the CPUID bit     */
        push    eax                 /* attempt to set EFLAGS to */
        popfd                       /*     the new value        */
```

```
        pushfd                      /* get the new EFLAGS value */
        pop     eax                 /*     into eax             */
        xor     eax,[esp]           /* xor with original value  */
        and     eax,edx             /* has CPUID bit changed?   */
        setne   al                  /* set to 1 if we have been */
        mov     ret_value,al        /*     able to change it    */
        popfd                       /* restore original EFLAGS  */
    }
    return (int)ret_value;
}

INLINE int is_via_cpu(void)
{   char ret_value;
    __asm
    {   push    ebx
        xor     eax,eax             /* use CPUID to get vendor  */
        cpuid                       /* identity string          */
        xor     eax,eax             /* is it "CentaurHauls" ?   */
        sub     ebx,0x746e6543      /* 'Cent'                   */
        or      eax,ebx
        sub     edx,0x48727561      /* 'aurH'                   */
        or      eax,edx
        sub     ecx,0x736c7561      /* 'auls'                   */
        or      eax,ecx
        sete    al                  /* set to 1 if it is VIA ID */
        mov     dl,NEH_CPU_READ     /* mark CPU type as read    */
        or      dl,al               /* & store result in flags  */
        mov     [via_flags],dl      /* set VIA detected flag    */
        mov     ret_value,al        /*     able to change it    */
        pop     ebx
    }
    return (int)ret_value;
}

INLINE int read_via_flags(void)
{   char ret_value = 0;
    __asm
    {   mov     eax,0xC0000000      /* Centaur extended CPUID   */
        cpuid
        mov     edx,0xc0000001      /* >= 0xc0000001 if support */
        cmp     eax,edx             /* for VIA extended feature */
        jnae    no_rng              /*     flags is available   */
        mov     eax,edx             /* read Centaur extended    */
        cpuid                       /*     feature flags        */
        mov     eax,NEH_FLAGS_MASK   /* mask out and save       */
        and     eax,edx             /*  the RNG and ACE flags   */
        or      [via_flags],al      /* present & enabled flags  */
        mov     ret_value,al        /*     able to change it    */
no_rng:
```

```
    }
    return (int)ret_value;
}

INLINE unsigned int via_rng_in(void *buf)
{   char ret_value = 0x1f;
    __asm
    {   push    edi
        mov     edi,buf         /* input buffer address     */
        xor     edx,edx         /* try to fetch 8 bytes     */
        NEH_RNG                 /* do RNG read operation    */
        and     ret_value,al    /* count of bytes returned  */
        pop     edi
    }
    return (int)ret_value;
}

INLINE void via_ecb_op5(
            const void *k, const void *c, const void *s, void *d, int
↪ l)
{   __asm
    {   push    ebx
        NEH_REKEY
        mov     ebx, (k)
        mov     edx, (c)
        mov     esi, (s)
        mov     edi, (d)
        mov     ecx, (l)
        NEH_ECB
        pop     ebx
    }
}

INLINE void via_cbc_op6(
            const void *k, const void *c, const void *s, void *d, int
↪ l, void *v)
{   __asm
    {   push    ebx
        NEH_REKEY
        mov     ebx, (k)
        mov     edx, (c)
        mov     esi, (s)
        mov     edi, (d)
        mov     ecx, (l)
        mov     eax, (v)
        NEH_CBC
        pop     ebx
    }
}
```

```
INLINE void via_cbc_op7(
        const void *k, const void *c, const void *s, void *d, int l,
↪void *v, void *w)
{   __asm
    {   push    ebx
        NEH_REKEY
        mov     ebx, (k)
        mov     edx, (c)
        mov     esi, (s)
        mov     edi, (d)
        mov     ecx, (l)
        mov     eax, (v)
        NEH_CBC
        mov     esi, eax
        mov     edi, (w)
        movsd
        movsd
        movsd
        movsd
        pop     ebx
    }
}

INLINE void via_cfb_op6(
            const void *k, const void *c, const void *s, void *d, int
↪ l, void *v)
{   __asm
    {   push    ebx
        NEH_REKEY
        mov     ebx, (k)
        mov     edx, (c)
        mov     esi, (s)
        mov     edi, (d)
        mov     ecx, (l)
        mov     eax, (v)
        NEH_CFB
        pop     ebx
    }
}

INLINE void via_cfb_op7(
        const void *k, const void *c, const void *s, void *d, int l,
↪void *v, void *w)
{   __asm
    {   push    ebx
        NEH_REKEY
        mov     ebx, (k)
        mov     edx, (c)
```

```
        mov     esi, (s)
        mov     edi, (d)
        mov     ecx, (l)
        mov     eax, (v)
        NEH_CFB
        mov     esi, eax
        mov     edi, (w)
        movsd
        movsd
        movsd
        movsd
        pop     ebx
    }
}

INLINE void via_ofb_op6(
            const void *k, const void *c, const void *s, void *d, int
↪ l, void *v)
{   __asm
    {   push    ebx
        NEH_REKEY
        mov     ebx, (k)
        mov     edx, (c)
        mov     esi, (s)
        mov     edi, (d)
        mov     ecx, (l)
        mov     eax, (v)
        NEH_OFB
        pop     ebx
    }
}

#elif defined( __GNUC__ )

#define NEH_REKEY   asm("pushfl\n popfl\n\t")
#define NEH_ECB     asm(".byte 0xf3, 0x0f, 0xa7, 0xc8\n\t")
#define NEH_CBC     asm(".byte 0xf3, 0x0f, 0xa7, 0xd0\n\t")
#define NEH_CFB     asm(".byte 0xf3, 0x0f, 0xa7, 0xe0\n\t")
#define NEH_OFB     asm(".byte 0xf3, 0x0f, 0xa7, 0xe8\n\t")
#define NEH_RNG     asm(".byte 0x0f, 0xa7, 0xc0\n\t");

INLINE int has_cpuid(void)
{   int val;
    asm("pushfl\n\t");
    asm("movl  0(%esp),%eax\n\t");
    asm("xor   $0x00200000,%eax\n\t");
    asm("pushl %eax\n\t");
    asm("popfl\n\t");
    asm("pushfl\n\t");
```

```
    asm("popl␣␣%eax\n\t");
    asm("xorl␣␣0(%esp),%edx\n\t");
    asm("andl␣␣$0x00200000,%eax\n\t");
    asm("movl␣␣%%eax,%0\n\t" : "=m" (val));
    asm("popfl\n\t");
    return val ? 1 : 0;
}

INLINE int is_via_cpu(void)
{   int val;
    asm("pushl␣%ebx\n\t");
    asm("xorl␣%eax,%eax\n\t");
    asm("cpuid\n\t");
    asm("xorl␣%eax,%eax\n\t");
    asm("subl␣$0x746e6543,%ebx\n\t");
    asm("orl␣␣%ebx,%eax\n\t");
    asm("subl␣$0x48727561,%edx\n\t");
    asm("orl␣␣%edx,%eax\n\t");
    asm("subl␣$0x736c7561,%ecx\n\t");
    asm("orl␣␣%ecx,%eax\n\t");
    asm("movl␣%%eax,%0\n\t" : "=m" (val));
    asm("popl␣%ebx\n\t");
    val = (val ? 0 : 1);
    via_flags = (val | NEH_CPU_READ);
    return val;
}

INLINE int read_via_flags(void)
{   unsigned char   val;
    asm("movl␣$0xc0000000,%eax\n\t");
    asm("cpuid\n\t");
    asm("movl␣$0xc0000001,%edx\n\t");
    asm("cmpl␣%edx,%eax\n\t");
    asm("setae␣%al\n\t");
    asm("movb␣%%al,%0\n\t" : "=m" (val));
    if(!val) return 0;
    asm("movl␣$0xc0000001,%eax\n\t");
    asm("cpuid\n\t");
    asm("movb␣%%dl,%0\n\t" : "=m" (val));
    val &= NEH_FLAGS_MASK;
    via_flags |= val;
    return (int) val;
}

INLINE int via_rng_in(void *buf)
{   int val;
    asm("pushl␣%edi\n\t");
    asm("movl␣%0,%%edi\n\t" : : "m" (buf));
    asm("xorl␣%edx,%edx\n\t");
```

```
    NEH_RNG
    asm("andl $0x0000001f,%eax\n\t");
    asm("movl %%eax,%0\n\t" : "=m" (val));
    asm("popl %edi\n\t");
    return val;
}

INLINE volatile  void via_ecb_op5(
            const void *k, const void *c, const void *s, void *d, int l)
{
    asm("pushl %ebx\n\t");
    NEH_REKEY;
    asm("movl %0, %%ebx\n\t" : : "m" (k));
    asm("movl %0, %%edx\n\t" : : "m" (c));
    asm("movl %0, %%esi\n\t" : : "m" (s));
    asm("movl %0, %%edi\n\t" : : "m" (d));
    asm("movl %0, %%ecx\n\t" : : "m" (l));
    NEH_ECB;
    asm("popl %ebx\n\t");
}

INLINE volatile  void via_cbc_op6(
            const void *k, const void *c, const void *s, void *d, int l, void *v)
{
    asm("pushl %ebx\n\t");
    NEH_REKEY;
    asm("movl %0, %%ebx\n\t" : : "m" (k));
    asm("movl %0, %%edx\n\t" : : "m" (c));
    asm("movl %0, %%esi\n\t" : : "m" (s));
    asm("movl %0, %%edi\n\t" : : "m" (d));
    asm("movl %0, %%ecx\n\t" : : "m" (l));
    asm("movl %0, %%eax\n\t" : : "m" (v));
    NEH_CBC;
    asm("popl %ebx\n\t");
}

INLINE volatile  void via_cbc_op7(
        const void *k, const void *c, const void *s, void *d, int l, void *v, void *w)
{
    asm("pushl %ebx\n\t");
    NEH_REKEY;
    asm("movl %0, %%ebx\n\t" : : "m" (k));
    asm("movl %0, %%edx\n\t" : : "m" (c));
    asm("movl %0, %%esi\n\t" : : "m" (s));
    asm("movl %0, %%edi\n\t" : : "m" (d));
    asm("movl %0, %%ecx\n\t" : : "m" (l));
```

```
    asm("movl␣%0,␣%%eax\n\t" : : "m" (v));
    NEH_CBC;
    asm("movl␣%eax,%esi\n\t");
    asm("movl␣%0,␣%%edi\n\t" : : "m" (w));
    asm("movsl;␣movsl;␣movsl;␣movsl\n\t");
    asm("popl␣%ebx\n\t");
}

INLINE volatile  void via_cfb_op6(
            const void *k, const void *c, const void *s, void *d, int
↪ l, void *v)
{
    asm("pushl␣%ebx\n\t");
    NEH_REKEY;
    asm("movl␣%0,␣%%ebx\n\t" : : "m" (k));
    asm("movl␣%0,␣%%edx\n\t" : : "m" (c));
    asm("movl␣%0,␣%%esi\n\t" : : "m" (s));
    asm("movl␣%0,␣%%edi\n\t" : : "m" (d));
    asm("movl␣%0,␣%%ecx\n\t" : : "m" (l));
    asm("movl␣%0,␣%%eax\n\t" : : "m" (v));
    NEH_CFB;
    asm("popl␣%ebx\n\t");
}

INLINE volatile  void via_cfb_op7(
        const void *k, const void *c, const void *s, void *d, int l,
↪void *v, void *w)
{
    asm("pushl␣%ebx\n\t");
    NEH_REKEY;
    asm("movl␣%0,␣%%ebx\n\t" : : "m" (k));
    asm("movl␣%0,␣%%edx\n\t" : : "m" (c));
    asm("movl␣%0,␣%%esi\n\t" : : "m" (s));
    asm("movl␣%0,␣%%edi\n\t" : : "m" (d));
    asm("movl␣%0,␣%%ecx\n\t" : : "m" (l));
    asm("movl␣%0,␣%%eax\n\t" : : "m" (v));
    NEH_CFB;
    asm("movl␣%eax,%esi\n\t");
    asm("movl␣%0,␣%%edi\n\t" : : "m" (w));
    asm("movsl;␣movsl;␣movsl;␣movsl\n\t");
    asm("popl␣%ebx\n\t");
}

INLINE volatile  void via_ofb_op6(
            const void *k, const void *c, const void *s, void *d, int
↪ l, void *v)
{
    asm("pushl␣%ebx\n\t");
    NEH_REKEY;
```

```c
    asm("movl %0, %%ebx\n\t" : : "m" (k));
    asm("movl %0, %%edx\n\t" : : "m" (c));
    asm("movl %0, %%esi\n\t" : : "m" (s));
    asm("movl %0, %%edi\n\t" : : "m" (d));
    asm("movl %0, %%ecx\n\t" : : "m" (l));
    asm("movl %0, %%eax\n\t" : : "m" (v));
    NEH_OFB;
    asm("popl %ebx\n\t");
}

#else
#error VIA ACE is not available with this compiler
#endif

INLINE int via_ace_test(void)
{
    return has_cpuid() && is_via_cpu() && ((read_via_flags() & NEH_ACE_FLAGS) == NEH_ACE_FLAGS);
}

#define VIA_ACE_AVAILABLE   (((via_flags & NEH_ACE_FLAGS) == NEH_ACE_FLAGS)         \
    || (via_flags & NEH_CPU_READ) && (via_flags & NEH_CPU_IS_VIA) || via_ace_test())

INLINE int via_rng_test(void)
{
    return has_cpuid() && is_via_cpu() && ((read_via_flags() & NEH_RNG_FLAGS) == NEH_RNG_FLAGS);
}

#define VIA_RNG_AVAILABLE   (((via_flags & NEH_RNG_FLAGS) == NEH_RNG_FLAGS)         \
    || (via_flags & NEH_CPU_READ) && (via_flags & NEH_CPU_IS_VIA) || via_rng_test())

INLINE int read_via_rng(void *buf, int count)
{   int nbr, max_reads, lcnt = count;
    unsigned char *p, *q;
    aligned_auto(unsigned char, bp, 64, 16);

    if(!VIA_RNG_AVAILABLE)
        return 0;

    do
    {
        max_reads = MAX_READ_ATTEMPTS;
        do
            nbr = via_rng_in(bp);
```

```
        while
            (nbr == 0 && --max_reads);

        lcnt -= nbr;
        p = (unsigned char*)buf; q = bp;
        while(nbr--)
            *p++ = *q++;
    }
    while
        (lcnt && max_reads);

    return count - lcnt;
}

#endif
```

src/Obf/aescrypt.c

```
/*
---------------------------------------------------------------------------
↪
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights
↪reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied
↪warranties
in respect of its operation, including, but not limited to,
↪correctness
and fitness for purpose.
---------------------------------------------------------------------------
↪
Issue Date: 20/12/2007
*/

#include "aesopt.h"
#include "aestab.h"

#if defined(__cplusplus)
extern "C"
{
#endif
```

```
#define si(y,x,k,c) (s(y,c) = word_in(x, c) ^ (k)[c])
#define so(y,x,c)   word_out(y, c, s(x,c))

#if defined(ARRAYS)
#define locals(y,x)     x[4],y[4]
#else
#define locals(y,x)     x##0,x##1,x##2,x##3,y##0,y##1,y##2,y##3
#endif

#define l_copy(y, x)    s(y,0) = s(x,0); s(y,1) = s(x,1); \
                        s(y,2) = s(x,2); s(y,3) = s(x,3);
#define state_in(y,x,k) si(y,x,k,0); si(y,x,k,1); si(y,x,k,2); si(y,x,k,3)
#define state_out(y,x)  so(y,x,0); so(y,x,1); so(y,x,2); so(y,x,3)
#define round(rm,y,x,k) rm(y,x,k,0); rm(y,x,k,1); rm(y,x,k,2); rm(y,x,k,3)

#if ( FUNCS_IN_C & ENCRYPTION_IN_C )

/* Visual C++ .Net v7.1 provides the fastest encryption code when using
   Pentium optimiation with small code but this is poor for decryption
   so we need to control this with the following VC++ pragmas
*/

#if defined( _MSC_VER ) && !defined( _WIN64 )
#pragma optimize( "s", on )
#endif

/* Given the column (c) of the output state variable, the following
   macros give the input state variables which are needed in its
   computation for each row (r) of the state. All the alternative
   macros give the same end values but expand into different ways
   of calculating these values.  In particular the complex macro
   used for dynamically variable block sizes is designed to expand
   to a compile time constant whenever possible but will expand to
   conditional clauses on some branches (I am grateful to Frank
   Yellin for this construction)
*/

#define fwd_var(x,r,c)\
 ( r == 0 ? ( c == 0 ? s(x,0) : c == 1 ? s(x,1) : c == 2 ? s(x,2) : s(x,3))\
 : r == 1 ? ( c == 0 ? s(x,1) : c == 1 ? s(x,2) : c == 2 ? s(x,3) : s(x,0))\
 : r == 2 ? ( c == 0 ? s(x,2) : c == 1 ? s(x,3) : c == 2 ? s(x,0) : s(x,1))\
```

```
 :             ( c == 0 ? s(x,3) : c == 1 ? s(x,0) : c == 2 ? s(x,1) : s(x,2)))

#if defined(FT4_SET)
#undef  dec_fmvars
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ four_tables(x,t_use(f,n),fwd_var,rf1,c))
#elif defined(FT1_SET)
#undef  dec_fmvars
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ one_table(x,upr,t_use(f,n),fwd_var,rf1,c))
#else
#define fwd_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ fwd_mcol(no_table(x,t_use(s,box),fwd_var,rf1,c)))
#endif

#if defined(FL4_SET)
#define fwd_lrnd(y,x,k,c)   (s(y,c) = (k)[c] ^ four_tables(x,t_use(f,l),fwd_var,rf1,c))
#elif defined(FL1_SET)
#define fwd_lrnd(y,x,k,c)   (s(y,c) = (k)[c] ^ one_table(x,ups,t_use(f,l),fwd_var,rf1,c))
#else
#define fwd_lrnd(y,x,k,c)   (s(y,c) = (k)[c] ^ no_table(x,t_use(s,box),fwd_var,rf1,c))
#endif

AES_RETURN aes_encrypt(const unsigned char *in, unsigned char *out, const aes_encrypt_ctx cx[1])
{   uint_32t         locals(b0, b1);
    const uint_32t   *kp;
#if defined( dec_fmvars )
    dec_fmvars; /* declare variables for fwd_mcol() if needed */
#endif

    if( cx->inf.b[0] != 10 * 16 && cx->inf.b[0] != 12 * 16 && cx->inf.b[0] != 14 * 16 )
        return EXIT_FAILURE;

    kp = cx->ks;
    state_in(b0, in, kp);

#if (ENC_UNROLL == FULL)

    switch(cx->inf.b[0])
    {
    case 14 * 16:
        round(fwd_rnd,  b1, b0, kp + 1 * N_COLS);
        round(fwd_rnd,  b0, b1, kp + 2 * N_COLS);
```

```
            kp += 2 * N_COLS;
        case 12 * 16:
            round(fwd_rnd,  b1, b0, kp + 1 * N_COLS);
            round(fwd_rnd,  b0, b1, kp + 2 * N_COLS);
            kp += 2 * N_COLS;
        case 10 * 16:
            round(fwd_rnd,  b1, b0, kp + 1 * N_COLS);
            round(fwd_rnd,  b0, b1, kp + 2 * N_COLS);
            round(fwd_rnd,  b1, b0, kp + 3 * N_COLS);
            round(fwd_rnd,  b0, b1, kp + 4 * N_COLS);
            round(fwd_rnd,  b1, b0, kp + 5 * N_COLS);
            round(fwd_rnd,  b0, b1, kp + 6 * N_COLS);
            round(fwd_rnd,  b1, b0, kp + 7 * N_COLS);
            round(fwd_rnd,  b0, b1, kp + 8 * N_COLS);
            round(fwd_rnd,  b1, b0, kp + 9 * N_COLS);
            round(fwd_lrnd, b0, b1, kp +10 * N_COLS);
    }

#else

#if (ENC_UNROLL == PARTIAL)
    {   uint_32t    rnd;
        for(rnd = 0; rnd < (cx->inf.b[0] >> 5) - 1; ++rnd)
        {
            kp += N_COLS;
            round(fwd_rnd, b1, b0, kp);
            kp += N_COLS;
            round(fwd_rnd, b0, b1, kp);
        }
        kp += N_COLS;
        round(fwd_rnd,  b1, b0, kp);
#else
    {   uint_32t    rnd;
        for(rnd = 0; rnd < (cx->inf.b[0] >> 4) - 1; ++rnd)
        {
            kp += N_COLS;
            round(fwd_rnd, b1, b0, kp);
            l_copy(b0, b1);
        }
#endif
        kp += N_COLS;
        round(fwd_lrnd, b0, b1, kp);
    }
#endif

    state_out(out, b0);
    return EXIT_SUCCESS;
}
```

```
#endif

#if ( FUNCS_IN_C & DECRYPTION_IN_C)

/* Visual C++ .Net v7.1 provides the fastest encryption code when using
   Pentium optimiation with small code but this is poor for decryption
   so we need to control this with the following VC++ pragmas
*/

#if defined( _MSC_VER ) && !defined( _WIN64 )
#pragma optimize( "t", on )
#endif

/* Given the column (c) of the output state variable, the following
   macros give the input state variables which are needed in its
   computation for each row (r) of the state. All the alternative
   macros give the same end values but expand into different ways
   of calculating these values.  In particular the complex macro
   used for dynamically variable block sizes is designed to expand
   to a compile time constant whenever possible but will expand to
   conditional clauses on some branches (I am grateful to Frank
   Yellin for this construction)
*/

#define inv_var(x,r,c)\
 ( r == 0 ? ( c == 0 ? s(x,0) : c == 1 ? s(x,1) : c == 2 ? s(x,2) : s(x,3))\
 : r == 1 ? ( c == 0 ? s(x,3) : c == 1 ? s(x,0) : c == 2 ? s(x,1) : s(x,2))\
 : r == 2 ? ( c == 0 ? s(x,2) : c == 1 ? s(x,3) : c == 2 ? s(x,0) : s(x,1))\
 :          ( c == 0 ? s(x,1) : c == 1 ? s(x,2) : c == 2 ? s(x,3) : s(x,0)))

#if defined(IT4_SET)
#undef  dec_imvars
#define inv_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ four_tables(x,t_use(i,n),inv_var,rf1,c))
#elif defined(IT1_SET)
#undef  dec_imvars
#define inv_rnd(y,x,k,c)    (s(y,c) = (k)[c] ^ one_table(x,upr,t_use(i,n),inv_var,rf1,c))
#else
#define inv_rnd(y,x,k,c)    (s(y,c) = inv_mcol((k)[c] ^ no_table(x,t_use(i,box),inv_var,rf1,c)))
#endif

```

```c
#if defined(IL4_SET)
#define inv_lrnd(y,x,k,c)   (s(y,c) = (k)[c] ^ four_tables(x,t_use(i,
↪l),inv_var,rf1,c))
#elif defined(IL1_SET)
#define inv_lrnd(y,x,k,c)   (s(y,c) = (k)[c] ^ one_table(x,ups,t_use(
↪i,l),inv_var,rf1,c))
#else
#define inv_lrnd(y,x,k,c)   (s(y,c) = (k)[c] ^ no_table(x,t_use(i,box
↪),inv_var,rf1,c))
#endif

/* This code can work with the decryption key schedule in the   */
/* order that is used for encrytpion (where the 1st decryption  */
/* round key is at the high end ot the schedule) or with a key  */
/* schedule that has been reversed to put the 1st decryption    */
/* round key at the low end of the schedule in memory (when     */
/* AES_REV_DKS is defined)                                      */

#ifdef AES_REV_DKS
#define key_ofs     0
#define rnd_key(n)  (kp + n * N_COLS)
#else
#define key_ofs     1
#define rnd_key(n)  (kp - n * N_COLS)
#endif

AES_RETURN aes_decrypt(const unsigned char *in, unsigned char *out,
↪const aes_decrypt_ctx cx[1])
{   uint_32t        locals(b0, b1);
#if defined( dec_imvars )
    dec_imvars; /* declare variables for inv_mcol() if needed */
#endif
    const uint_32t *kp;

    if( cx->inf.b[0] != 10 * 16 && cx->inf.b[0] != 12 * 16 && cx->inf
↪.b[0] != 14 * 16 )
        return EXIT_FAILURE;

    kp = cx->ks + (key_ofs ? (cx->inf.b[0] >> 2) : 0);
    state_in(b0, in, kp);

#if (DEC_UNROLL == FULL)

    kp = cx->ks + (key_ofs ? 0 : (cx->inf.b[0] >> 2));
    switch(cx->inf.b[0])
    {
    case 14 * 16:
        round(inv_rnd,  b1, b0, rnd_key(-13));
        round(inv_rnd,  b0, b1, rnd_key(-12));
```

```
    case 12 * 16:
        round(inv_rnd,  b1, b0, rnd_key(-11));
        round(inv_rnd,  b0, b1, rnd_key(-10));
    case 10 * 16:
        round(inv_rnd,  b1, b0, rnd_key(-9));
        round(inv_rnd,  b0, b1, rnd_key(-8));
        round(inv_rnd,  b1, b0, rnd_key(-7));
        round(inv_rnd,  b0, b1, rnd_key(-6));
        round(inv_rnd,  b1, b0, rnd_key(-5));
        round(inv_rnd,  b0, b1, rnd_key(-4));
        round(inv_rnd,  b1, b0, rnd_key(-3));
        round(inv_rnd,  b0, b1, rnd_key(-2));
        round(inv_rnd,  b1, b0, rnd_key(-1));
        round(inv_lrnd, b0, b1, rnd_key( 0));
    }

#else

#if (DEC_UNROLL == PARTIAL)
    {   uint_32t    rnd;
        for(rnd = 0; rnd < (cx->inf.b[0] >> 5) - 1; ++rnd)
        {
            kp = rnd_key(1);
            round(inv_rnd, b1, b0, kp);
            kp = rnd_key(1);
            round(inv_rnd, b0, b1, kp);
        }
        kp = rnd_key(1);
        round(inv_rnd, b1, b0, kp);
#else
    {   uint_32t    rnd;
        for(rnd = 0; rnd < (cx->inf.b[0] >> 4) - 1; ++rnd)
        {
            kp = rnd_key(1);
            round(inv_rnd, b1, b0, kp);
            l_copy(b0, b1);
        }
#endif
        kp = rnd_key(1);
        round(inv_lrnd, b0, b1, kp);
        }
#endif

    state_out(out, b0);
    return EXIT_SUCCESS;
}

#endif
```

```c
#if defined(__cplusplus)
}
#endif
```

src/Obf/aeskey.c

```c
/*
---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 20/12/2007
*/

#include "aesopt.h"
#include "aestab.h"

#ifdef USE_VIA_ACE_IF_PRESENT
#  include "aes_via_ace.h"
#endif

#if defined(__cplusplus)
extern "C"
{
#endif

/* Initialise the key schedule from the user supplied key. The key
   length can be specified in bytes, with legal values of 16, 24
   and 32, or in bits, with legal values of 128, 192 and 256. These
   values correspond with Nk values of 4, 6 and 8 respectively.

   The following macros implement a single cycle in the key
   schedule generation process. The number of cycles needed
```

```
    for each cx->n_col and nk value is:

     nk =             4   5   6   7   8
     ------------------------------
     cx->n_col = 4   10   9   8   7   7
     cx->n_col = 5   14  11  10   9   9
     cx->n_col = 6   19  15  12  11  11
     cx->n_col = 7   21  19  16  13  14
     cx->n_col = 8   29  23  19  17  14
*/

#if defined( REDUCE_CODE_SIZE )
#   define ls_box ls_sub
    uint_32t ls_sub(const uint_32t t, const uint_32t n);
#   define inv_mcol im_sub
    uint_32t im_sub(const uint_32t x);
#   ifdef ENC_KS_UNROLL
#     undef ENC_KS_UNROLL
#   endif
#   ifdef DEC_KS_UNROLL
#     undef DEC_KS_UNROLL
#   endif
#endif

#if (FUNCS_IN_C & ENC_KEYING_IN_C)

#if defined(AES_128) || defined( AES_VAR )

#define ke4(k,i) \
{   k[4*(i)+4] = ss[0] ^= ls_box(ss[3],3) ^ t_use(r,c)[i]; \
    k[4*(i)+5] = ss[1] ^= ss[0]; \
    k[4*(i)+6] = ss[2] ^= ss[1]; \
    k[4*(i)+7] = ss[3] ^= ss[2]; \
}

AES_RETURN aes_encrypt_key128(const unsigned char *key,
↪aes_encrypt_ctx cx[1])
{   uint_32t    ss[4];

    cx->ks[0] = ss[0] = word_in(key, 0);
    cx->ks[1] = ss[1] = word_in(key, 1);
    cx->ks[2] = ss[2] = word_in(key, 2);
    cx->ks[3] = ss[3] = word_in(key, 3);

#ifdef ENC_KS_UNROLL
    ke4(cx->ks, 0);  ke4(cx->ks, 1);
    ke4(cx->ks, 2);  ke4(cx->ks, 3);
    ke4(cx->ks, 4);  ke4(cx->ks, 5);
    ke4(cx->ks, 6);  ke4(cx->ks, 7);
```

```
    ke4(cx->ks, 8);
#else
    {   uint_32t i;
        for(i = 0; i < 9; ++i)
            ke4(cx->ks, i);
    }
#endif
    ke4(cx->ks, 9);
    cx->inf.l = 0;
    cx->inf.b[0] = 10 * 16;

#ifdef USE_VIA_ACE_IF_PRESENT
    if(VIA_ACE_AVAILABLE)
        cx->inf.b[1] = 0xff;
#endif
    return EXIT_SUCCESS;
}

#endif

#if defined(AES_192) || defined( AES_VAR )

#define kef6(k,i) \
{   k[6*(i)+ 6] = ss[0] ^= ls_box(ss[5],3) ^ t_use(r,c)[i]; \
    k[6*(i)+ 7] = ss[1] ^= ss[0]; \
    k[6*(i)+ 8] = ss[2] ^= ss[1]; \
    k[6*(i)+ 9] = ss[3] ^= ss[2]; \
}

#define ke6(k,i) \
{   kef6(k,i); \
    k[6*(i)+10] = ss[4] ^= ss[3]; \
    k[6*(i)+11] = ss[5] ^= ss[4]; \
}

AES_RETURN aes_encrypt_key192(const unsigned char *key,
↪aes_encrypt_ctx cx[1])
{   uint_32t    ss[6];

    cx->ks[0] = ss[0] = word_in(key, 0);
    cx->ks[1] = ss[1] = word_in(key, 1);
    cx->ks[2] = ss[2] = word_in(key, 2);
    cx->ks[3] = ss[3] = word_in(key, 3);
    cx->ks[4] = ss[4] = word_in(key, 4);
    cx->ks[5] = ss[5] = word_in(key, 5);

#ifdef ENC_KS_UNROLL
    ke6(cx->ks, 0);  ke6(cx->ks, 1);
    ke6(cx->ks, 2);  ke6(cx->ks, 3);
```

```
    ke6(cx->ks, 4);  ke6(cx->ks, 5);
    ke6(cx->ks, 6);
#else
    {   uint_32t i;
        for(i = 0; i < 7; ++i)
            ke6(cx->ks, i);
    }
#endif
    kef6(cx->ks, 7);
    cx->inf.l = 0;
    cx->inf.b[0] = 12 * 16;

#ifdef USE_VIA_ACE_IF_PRESENT
    if(VIA_ACE_AVAILABLE)
        cx->inf.b[1] = 0xff;
#endif
    return EXIT_SUCCESS;
}

#endif

#if defined(AES_256) || defined( AES_VAR )

#define kef8(k,i) \
{   k[8*(i)+ 8] = ss[0] ^= ls_box(ss[7],3) ^ t_use(r,c)[i]; \
    k[8*(i)+ 9] = ss[1] ^= ss[0]; \
    k[8*(i)+10] = ss[2] ^= ss[1]; \
    k[8*(i)+11] = ss[3] ^= ss[2]; \
}

#define ke8(k,i) \
{   kef8(k,i); \
    k[8*(i)+12] = ss[4] ^= ls_box(ss[3],0); \
    k[8*(i)+13] = ss[5] ^= ss[4]; \
    k[8*(i)+14] = ss[6] ^= ss[5]; \
    k[8*(i)+15] = ss[7] ^= ss[6]; \
}

AES_RETURN aes_encrypt_key256(const unsigned char *key,
↪aes_encrypt_ctx cx[1])
{   uint_32t    ss[8];

    cx->ks[0] = ss[0] = word_in(key, 0);
    cx->ks[1] = ss[1] = word_in(key, 1);
    cx->ks[2] = ss[2] = word_in(key, 2);
    cx->ks[3] = ss[3] = word_in(key, 3);
    cx->ks[4] = ss[4] = word_in(key, 4);
    cx->ks[5] = ss[5] = word_in(key, 5);
    cx->ks[6] = ss[6] = word_in(key, 6);
```

```
    cx->ks[7] = ss[7] = word_in(key, 7);

#ifdef ENC_KS_UNROLL
    ke8(cx->ks, 0); ke8(cx->ks, 1);
    ke8(cx->ks, 2); ke8(cx->ks, 3);
    ke8(cx->ks, 4); ke8(cx->ks, 5);
#else
    {   uint_32t i;
        for(i = 0; i < 6; ++i)
            ke8(cx->ks,  i);
    }
#endif
    kef8(cx->ks, 6);
    cx->inf.l = 0;
    cx->inf.b[0] = 14 * 16;

#ifdef USE_VIA_ACE_IF_PRESENT
    if(VIA_ACE_AVAILABLE)
        cx->inf.b[1] = 0xff;
#endif
    return EXIT_SUCCESS;
}

#endif

#if defined( AES_VAR )

AES_RETURN aes_encrypt_key(const unsigned char *key, int key_len,
↪aes_encrypt_ctx cx[1])
{
    switch(key_len)
    {
    case 16: case 128: return aes_encrypt_key128(key, cx);
    case 24: case 192: return aes_encrypt_key192(key, cx);
    case 32: case 256: return aes_encrypt_key256(key, cx);
    default: return EXIT_FAILURE;
    }
}

#endif

#endif

#if (FUNCS_IN_C & DEC_KEYING_IN_C)

/* this is used to store the decryption round keys   */
/* in forward or reverse order                       */

#ifdef AES_REV_DKS
```

```
#define v(n,i)  ((n) - (i) + 2 * ((i) & 3))
#else
#define v(n,i)  (i)
#endif

#if DEC_ROUND == NO_TABLES
#define ff(x)   (x)
#else
#define ff(x)   inv_mcol(x)
#if defined( dec_imvars )
#define d_vars  dec_imvars
#endif
#endif

#if defined(AES_128) || defined( AES_VAR )

#define k4e(k,i) \
{   k[v(40,(4*(i))+4)] = ss[0] ^= ls_box(ss[3],3) ^ t_use(r,c)[i]; \
    k[v(40,(4*(i))+5)] = ss[1] ^= ss[0]; \
    k[v(40,(4*(i))+6)] = ss[2] ^= ss[1]; \
    k[v(40,(4*(i))+7)] = ss[3] ^= ss[2]; \
}

#if 1

#define kdf4(k,i) \
{   ss[0] = ss[0] ^ ss[2] ^ ss[1] ^ ss[3]; \
    ss[1] = ss[1] ^ ss[3]; \
    ss[2] = ss[2] ^ ss[3]; \
    ss[4] = ls_box(ss[(i+3) % 4], 3) ^ t_use(r,c)[i]; \
    ss[i % 4] ^= ss[4]; \
    ss[4] ^= k[v(40,(4*(i)))];   k[v(40,(4*(i))+4)] = ff(ss[4]); \
    ss[4] ^= k[v(40,(4*(i))+1)]; k[v(40,(4*(i))+5)] = ff(ss[4]); \
    ss[4] ^= k[v(40,(4*(i))+2)]; k[v(40,(4*(i))+6)] = ff(ss[4]); \
    ss[4] ^= k[v(40,(4*(i))+3)]; k[v(40,(4*(i))+7)] = ff(ss[4]); \
}

#define kd4(k,i) \
{   ss[4] = ls_box(ss[(i+3) % 4], 3) ^ t_use(r,c)[i]; \
    ss[i % 4] ^= ss[4]; ss[4] = ff(ss[4]); \
    k[v(40,(4*(i))+4)] = ss[4] ^= k[v(40,(4*(i)))]; \
    k[v(40,(4*(i))+5)] = ss[4] ^= k[v(40,(4*(i))+1)]; \
    k[v(40,(4*(i))+6)] = ss[4] ^= k[v(40,(4*(i))+2)]; \
    k[v(40,(4*(i))+7)] = ss[4] ^= k[v(40,(4*(i))+3)]; \
}

#define kdl4(k,i) \
{   ss[4] = ls_box(ss[(i+3) % 4], 3) ^ t_use(r,c)[i]; ss[i % 4] ^= ss
↪[4]; \
```

```
    k[v(40,(4*(i))+4)] = (ss[0] ^= ss[1]) ^ ss[2] ^ ss[3]; \
    k[v(40,(4*(i))+5)] = ss[1] ^ ss[3]; \
    k[v(40,(4*(i))+6)] = ss[0]; \
    k[v(40,(4*(i))+7)] = ss[1]; \
}

#else

#define kdf4(k,i) \
{   ss[0] ^= ls_box(ss[3],3) ^ t_use(r,c)[i]; k[v(40,(4*(i))+ 4)] =
↪ff(ss[0]); \
    ss[1] ^= ss[0]; k[v(40,(4*(i))+ 5)] = ff(ss[1]); \
    ss[2] ^= ss[1]; k[v(40,(4*(i))+ 6)] = ff(ss[2]); \
    ss[3] ^= ss[2]; k[v(40,(4*(i))+ 7)] = ff(ss[3]); \
}

#define kd4(k,i) \
{   ss[4] = ls_box(ss[3],3) ^ t_use(r,c)[i]; \
    ss[0] ^= ss[4]; ss[4] = ff(ss[4]); k[v(40,(4*(i))+ 4)] = ss[4] ^=
↪ k[v(40,(4*(i)))]; \
    ss[1] ^= ss[0]; k[v(40,(4*(i))+ 5)] = ss[4] ^= k[v(40,(4*(i))+ 1)
↪]; \
    ss[2] ^= ss[1]; k[v(40,(4*(i))+ 6)] = ss[4] ^= k[v(40,(4*(i))+ 2)
↪]; \
    ss[3] ^= ss[2]; k[v(40,(4*(i))+ 7)] = ss[4] ^= k[v(40,(4*(i))+ 3)
↪]; \
}

#define kdl4(k,i) \
{   ss[0] ^= ls_box(ss[3],3) ^ t_use(r,c)[i]; k[v(40,(4*(i))+ 4)] =
↪ss[0]; \
    ss[1] ^= ss[0]; k[v(40,(4*(i))+ 5)] = ss[1]; \
    ss[2] ^= ss[1]; k[v(40,(4*(i))+ 6)] = ss[2]; \
    ss[3] ^= ss[2]; k[v(40,(4*(i))+ 7)] = ss[3]; \
}

#endif

AES_RETURN aes_decrypt_key128(const unsigned char *key,
↪aes_decrypt_ctx cx[1])
{   uint_32t    ss[5];
#if defined( d_vars )
        d_vars;
#endif
    cx->ks[v(40,(0))] = ss[0] = word_in(key, 0);
    cx->ks[v(40,(1))] = ss[1] = word_in(key, 1);
    cx->ks[v(40,(2))] = ss[2] = word_in(key, 2);
    cx->ks[v(40,(3))] = ss[3] = word_in(key, 3);

```

```
#ifdef DEC_KS_UNROLL
     kdf4(cx->ks, 0); kd4(cx->ks, 1);
     kd4(cx->ks, 2);  kd4(cx->ks, 3);
     kd4(cx->ks, 4);  kd4(cx->ks, 5);
     kd4(cx->ks, 6);  kd4(cx->ks, 7);
     kd4(cx->ks, 8);  kdl4(cx->ks, 9);
#else
    {   uint_32t i;
        for(i = 0; i < 10; ++i)
            k4e(cx->ks, i);
#if !(DEC_ROUND == NO_TABLES)
        for(i = N_COLS; i < 10 * N_COLS; ++i)
            cx->ks[i] = inv_mcol(cx->ks[i]);
#endif
    }
#endif
    cx->inf.l = 0;
    cx->inf.b[0] = 10 * 16;

#ifdef USE_VIA_ACE_IF_PRESENT
    if(VIA_ACE_AVAILABLE)
        cx->inf.b[1] = 0xff;
#endif
    return EXIT_SUCCESS;
}

#endif

#if defined(AES_192) || defined( AES_VAR )

#define k6ef(k,i) \
{   k[v(48,(6*(i))+ 6)] = ss[0] ^= ls_box(ss[5],3) ^ t_use(r,c)[i]; \
    k[v(48,(6*(i))+ 7)] = ss[1] ^= ss[0]; \
    k[v(48,(6*(i))+ 8)] = ss[2] ^= ss[1]; \
    k[v(48,(6*(i))+ 9)] = ss[3] ^= ss[2]; \
}

#define k6e(k,i) \
{   k6ef(k,i); \
    k[v(48,(6*(i))+10)] = ss[4] ^= ss[3]; \
    k[v(48,(6*(i))+11)] = ss[5] ^= ss[4]; \
}

#define kdf6(k,i) \
{   ss[0] ^= ls_box(ss[5],3) ^ t_use(r,c)[i]; k[v(48,(6*(i))+ 6)] =
↪ff(ss[0]); \
    ss[1] ^= ss[0]; k[v(48,(6*(i))+ 7)] = ff(ss[1]); \
    ss[2] ^= ss[1]; k[v(48,(6*(i))+ 8)] = ff(ss[2]); \
    ss[3] ^= ss[2]; k[v(48,(6*(i))+ 9)] = ff(ss[3]); \
```

```
    ss[4] ^= ss[3]; k[v(48,(6*(i))+10)] = ff(ss[4]); \
    ss[5] ^= ss[4]; k[v(48,(6*(i))+11)] = ff(ss[5]); \
}

#define kd6(k,i) \
{   ss[6] = ls_box(ss[5],3) ^ t_use(r,c)[i]; \
    ss[0] ^= ss[6]; ss[6] = ff(ss[6]); k[v(48,(6*(i))+ 6)] = ss[6] ^=
↪ k[v(48,(6*(i)))]; \
    ss[1] ^= ss[0]; k[v(48,(6*(i))+ 7)] = ss[6] ^= k[v(48,(6*(i))+ 1)
↪]; \
    ss[2] ^= ss[1]; k[v(48,(6*(i))+ 8)] = ss[6] ^= k[v(48,(6*(i))+ 2)
↪]; \
    ss[3] ^= ss[2]; k[v(48,(6*(i))+ 9)] = ss[6] ^= k[v(48,(6*(i))+ 3)
↪]; \
    ss[4] ^= ss[3]; k[v(48,(6*(i))+10)] = ss[6] ^= k[v(48,(6*(i))+ 4)
↪]; \
    ss[5] ^= ss[4]; k[v(48,(6*(i))+11)] = ss[6] ^= k[v(48,(6*(i))+ 5)
↪]; \
}

#define kdl6(k,i) \
{   ss[0] ^= ls_box(ss[5],3) ^ t_use(r,c)[i]; k[v(48,(6*(i))+ 6)] =
↪ss[0]; \
    ss[1] ^= ss[0]; k[v(48,(6*(i))+ 7)] = ss[1]; \
    ss[2] ^= ss[1]; k[v(48,(6*(i))+ 8)] = ss[2]; \
    ss[3] ^= ss[2]; k[v(48,(6*(i))+ 9)] = ss[3]; \
}

AES_RETURN aes_decrypt_key192(const unsigned char *key,
↪aes_decrypt_ctx cx[1])
{   uint_32t    ss[7];
#if defined( d_vars )
        d_vars;
#endif
    cx->ks[v(48,(0))] = ss[0] = word_in(key, 0);
    cx->ks[v(48,(1))] = ss[1] = word_in(key, 1);
    cx->ks[v(48,(2))] = ss[2] = word_in(key, 2);
    cx->ks[v(48,(3))] = ss[3] = word_in(key, 3);

#ifdef DEC_KS_UNROLL
    cx->ks[v(48,(4))] = ff(ss[4] = word_in(key, 4));
    cx->ks[v(48,(5))] = ff(ss[5] = word_in(key, 5));
    kdf6(cx->ks, 0); kd6(cx->ks, 1);
    kd6(cx->ks, 2);  kd6(cx->ks, 3);
    kd6(cx->ks, 4);  kd6(cx->ks, 5);
    kd6(cx->ks, 6);  kdl6(cx->ks, 7);
#else
    cx->ks[v(48,(4))] = ss[4] = word_in(key, 4);
    cx->ks[v(48,(5))] = ss[5] = word_in(key, 5);
```

```
    {   uint_32t i;

        for(i = 0; i < 7; ++i)
            k6e(cx->ks, i);
        k6ef(cx->ks, 7);
#if !(DEC_ROUND == NO_TABLES)
        for(i = N_COLS; i < 12 * N_COLS; ++i)
            cx->ks[i] = inv_mcol(cx->ks[i]);
#endif
    }
#endif
    cx->inf.l = 0;
    cx->inf.b[0] = 12 * 16;

#ifdef USE_VIA_ACE_IF_PRESENT
    if(VIA_ACE_AVAILABLE)
        cx->inf.b[1] = 0xff;
#endif
    return EXIT_SUCCESS;
}

#endif

#if defined(AES_256) || defined( AES_VAR )

#define k8ef(k,i) \
{   k[v(56,(8*(i))+ 8)] = ss[0] ^= ls_box(ss[7],3) ^ t_use(r,c)[i]; \
    k[v(56,(8*(i))+ 9)] = ss[1] ^= ss[0]; \
    k[v(56,(8*(i))+10)] = ss[2] ^= ss[1]; \
    k[v(56,(8*(i))+11)] = ss[3] ^= ss[2]; \
}

#define k8e(k,i) \
{   k8ef(k,i); \
    k[v(56,(8*(i))+12)] = ss[4] ^= ls_box(ss[3],0); \
    k[v(56,(8*(i))+13)] = ss[5] ^= ss[4]; \
    k[v(56,(8*(i))+14)] = ss[6] ^= ss[5]; \
    k[v(56,(8*(i))+15)] = ss[7] ^= ss[6]; \
}

#define kdf8(k,i) \
{   ss[0] ^= ls_box(ss[7],3) ^ t_use(r,c)[i]; k[v(56,(8*(i))+ 8)] = ff(ss[0]); \
    ss[1] ^= ss[0]; k[v(56,(8*(i))+ 9)] = ff(ss[1]); \
    ss[2] ^= ss[1]; k[v(56,(8*(i))+10)] = ff(ss[2]); \
    ss[3] ^= ss[2]; k[v(56,(8*(i))+11)] = ff(ss[3]); \
    ss[4] ^= ls_box(ss[3],0); k[v(56,(8*(i))+12)] = ff(ss[4]); \
    ss[5] ^= ss[4]; k[v(56,(8*(i))+13)] = ff(ss[5]); \
    ss[6] ^= ss[5]; k[v(56,(8*(i))+14)] = ff(ss[6]); \
```

```c
    ss[7] ^= ss[6]; k[v(56,(8*(i))+15)] = ff(ss[7]); \
}

#define kd8(k,i) \
{   ss[8] = ls_box(ss[7],3) ^ t_use(r,c)[i]; \
    ss[0] ^= ss[8]; ss[8] = ff(ss[8]); k[v(56,(8*(i))+ 8)] = ss[8] ^= k[v(56,(8*(i)))]; \
    ss[1] ^= ss[0]; k[v(56,(8*(i))+ 9)] = ss[8] ^= k[v(56,(8*(i))+ 1)]; \
    ss[2] ^= ss[1]; k[v(56,(8*(i))+10)] = ss[8] ^= k[v(56,(8*(i))+ 2)]; \
    ss[3] ^= ss[2]; k[v(56,(8*(i))+11)] = ss[8] ^= k[v(56,(8*(i))+ 3)]; \
    ss[8] = ls_box(ss[3],0); \
    ss[4] ^= ss[8]; ss[8] = ff(ss[8]); k[v(56,(8*(i))+12)] = ss[8] ^= k[v(56,(8*(i))+ 4)]; \
    ss[5] ^= ss[4]; k[v(56,(8*(i))+13)] = ss[8] ^= k[v(56,(8*(i))+ 5)]; \
    ss[6] ^= ss[5]; k[v(56,(8*(i))+14)] = ss[8] ^= k[v(56,(8*(i))+ 6)]; \
    ss[7] ^= ss[6]; k[v(56,(8*(i))+15)] = ss[8] ^= k[v(56,(8*(i))+ 7)]; \
}

#define kdl8(k,i) \
{   ss[0] ^= ls_box(ss[7],3) ^ t_use(r,c)[i]; k[v(56,(8*(i))+ 8)] = ss[0]; \
    ss[1] ^= ss[0]; k[v(56,(8*(i))+ 9)] = ss[1]; \
    ss[2] ^= ss[1]; k[v(56,(8*(i))+10)] = ss[2]; \
    ss[3] ^= ss[2]; k[v(56,(8*(i))+11)] = ss[3]; \
}

AES_RETURN aes_decrypt_key256(const unsigned char *key, aes_decrypt_ctx cx[1])
{   uint_32t    ss[9];
#if defined( d_vars )
        d_vars;
#endif
    cx->ks[v(56,(0))] = ss[0] = word_in(key, 0);
    cx->ks[v(56,(1))] = ss[1] = word_in(key, 1);
    cx->ks[v(56,(2))] = ss[2] = word_in(key, 2);
    cx->ks[v(56,(3))] = ss[3] = word_in(key, 3);

#ifdef DEC_KS_UNROLL
    cx->ks[v(56,(4))] = ff(ss[4] = word_in(key, 4));
    cx->ks[v(56,(5))] = ff(ss[5] = word_in(key, 5));
    cx->ks[v(56,(6))] = ff(ss[6] = word_in(key, 6));
    cx->ks[v(56,(7))] = ff(ss[7] = word_in(key, 7));
    kdf8(cx->ks, 0); kd8(cx->ks, 1);
```

```
    kd8(cx->ks, 2);  kd8(cx->ks, 3);
    kd8(cx->ks, 4);  kd8(cx->ks, 5);
    kdl8(cx->ks, 6);
#else
    cx->ks[v(56,(4))] = ss[4] = word_in(key, 4);
    cx->ks[v(56,(5))] = ss[5] = word_in(key, 5);
    cx->ks[v(56,(6))] = ss[6] = word_in(key, 6);
    cx->ks[v(56,(7))] = ss[7] = word_in(key, 7);
    {   uint_32t i;

        for(i = 0; i < 6; ++i)
            k8e(cx->ks,  i);
        k8ef(cx->ks,  6);
#if !(DEC_ROUND == NO_TABLES)
        for(i = N_COLS; i < 14 * N_COLS; ++i)
            cx->ks[i] = inv_mcol(cx->ks[i]);
#endif
    }
#endif
    cx->inf.l = 0;
    cx->inf.b[0] = 14 * 16;

#ifdef USE_VIA_ACE_IF_PRESENT
    if(VIA_ACE_AVAILABLE)
        cx->inf.b[1] = 0xff;
#endif
    return EXIT_SUCCESS;
}

#endif

#if defined( AES_VAR )

AES_RETURN aes_decrypt_key(const unsigned char *key, int key_len,
↪aes_decrypt_ctx cx[1])
{
    switch(key_len)
    {
    case 16: case 128: return aes_decrypt_key128(key, cx);
    case 24: case 192: return aes_decrypt_key192(key, cx);
    case 32: case 256: return aes_decrypt_key256(key, cx);
    default: return EXIT_FAILURE;
    }
}

#endif

#endif
```

```
#if defined(__cplusplus)
}
#endif
```

src/Obf/aesopt.h

```
/*
---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 20/12/2007

 This file contains the compilation options for AES (Rijndael) and code
 that is common across encryption, key scheduling and table generation.

 OPERATION

 These source code files implement the AES algorithm Rijndael designed by
 Joan Daemen and Vincent Rijmen. This version is designed for the standard
 block size of 16 bytes and for key sizes of 128, 192 and 256 bits (16, 24
 and 32 bytes).

 This version is designed for flexibility and speed using operations on
 32-bit words rather than operations on bytes.  It can be compiled with
 either big or little endian internal byte order but is faster when
```

```
↪the
native byte order for the processor is used.

THE CIPHER INTERFACE

The cipher interface is implemented as an array of bytes in which
↪lower
AES bit sequence indexes map to higher numeric significance within
↪bytes.

  uint_8t                  (an unsigned  8-bit type)
  uint_32t                 (an unsigned 32-bit type)
  struct aes_encrypt_ctx   (structure for the cipher encryption
↪context)
  struct aes_decrypt_ctx   (structure for the cipher decryption
↪context)
  AES_RETURN                 the function return type

  C subroutine calls:

  AES_RETURN aes_encrypt_key128(const unsigned char *key,
↪aes_encrypt_ctx cx[1]);
  AES_RETURN aes_encrypt_key192(const unsigned char *key,
↪aes_encrypt_ctx cx[1]);
  AES_RETURN aes_encrypt_key256(const unsigned char *key,
↪aes_encrypt_ctx cx[1]);
  AES_RETURN aes_encrypt(const unsigned char *in, unsigned char *out,
                                                          const
↪aes_encrypt_ctx cx[1]);

  AES_RETURN aes_decrypt_key128(const unsigned char *key,
↪aes_decrypt_ctx cx[1]);
  AES_RETURN aes_decrypt_key192(const unsigned char *key,
↪aes_decrypt_ctx cx[1]);
  AES_RETURN aes_decrypt_key256(const unsigned char *key,
↪aes_decrypt_ctx cx[1]);
  AES_RETURN aes_decrypt(const unsigned char *in, unsigned char *out,
                                                          const
↪aes_decrypt_ctx cx[1]);

IMPORTANT NOTE: If you are using this C interface with dynamic
↪tables make sure that
you call aes_init() before AES is used so that the tables are
↪initialised.

C++ aes class subroutines:

    Class AESencrypt  for encryption

```

```
        Construtors:
            AESencrypt(void)
            AESencrypt(const unsigned char *key) - 128 bit key
        Members:
            AES_RETURN key128(const unsigned char *key)
            AES_RETURN key192(const unsigned char *key)
            AES_RETURN key256(const unsigned char *key)
            AES_RETURN encrypt(const unsigned char *in, unsigned char *out) const

        Class AESdecrypt  for encryption
        Construtors:
            AESdecrypt(void)
            AESdecrypt(const unsigned char *key) - 128 bit key
        Members:
            AES_RETURN key128(const unsigned char *key)
            AES_RETURN key192(const unsigned char *key)
            AES_RETURN key256(const unsigned char *key)
            AES_RETURN decrypt(const unsigned char *in, unsigned char *out) const
*/

#if !defined( _AESOPT_H )
#define _AESOPT_H

#if defined( __cplusplus )
#include "aescpp.h"
#else
#include "aes.h"
#endif

/*  PLATFORM SPECIFIC INCLUDES */

#include "brg_endian.h"

/*  CONFIGURATION - THE USE OF DEFINES

    Later in this section there are a number of defines that control the
    operation of the code.  In each section, the purpose of each define is
    explained so that the relevant form can be included or excluded by
    setting either 1's or 0's respectively on the branches of the related
    #if clauses.  The following local defines should not be changed.
*/

#define ENCRYPTION_IN_C     1
```

```c
#define DECRYPTION_IN_C      2
#define ENC_KEYING_IN_C      4
#define DEC_KEYING_IN_C      8

#define NO_TABLES            0
#define ONE_TABLE            1
#define FOUR_TABLES          4
#define NONE                 0
#define PARTIAL              1
#define FULL                 2

/*  --- START OF USER CONFIGURED OPTIONS --- */

/*  1. BYTE ORDER WITHIN 32 BIT WORDS

    The fundamental data processing units in Rijndael are 8-bit bytes. The
    input, output and key input are all enumerated arrays of bytes in which
    bytes are numbered starting at zero and increasing to one less than the
    number of bytes in the array in question. This enumeration is only used
    for naming bytes and does not imply any adjacency or order relationship
    from one byte to another. When these inputs and outputs are considered
    as bit sequences, bits 8*n to 8*n+7 of the bit sequence are mapped to
    byte[n] with bit 8n+i in the sequence mapped to bit 7-i within the byte.
    In this implementation bits are numbered from 0 to 7 starting at the
    numerically least significant end of each byte (bit n represents 2^n).

    However, Rijndael can be implemented more efficiently using 32-bit
    words by packing bytes into words so that bytes 4*n to 4*n+3 are placed
    into word[n]. While in principle these bytes can be assembled into words
    in any positions, this implementation only supports the two formats in
    which bytes in adjacent positions within words also have adjacent byte
    numbers. This order is called big-endian if the lowest numbered bytes
    in words have the highest numeric significance and little-endian
```

```
↪if the
    opposite applies.

    This code can work in either order irrespective of the order used
↪ by the
    machine on which it runs. Normally the internal byte order will
↪be set
    to the order of the processor on which the code is to be run but
↪this
    define can be used to reverse this in special situations

    WARNING: Assembler code versions rely on PLATFORM_BYTE_ORDER
↪being set.
    This define will hence be redefined later (in section 4) if
↪necessary
*/

#if 1
#  define ALGORITHM_BYTE_ORDER PLATFORM_BYTE_ORDER
#elif 0
#  define ALGORITHM_BYTE_ORDER IS_LITTLE_ENDIAN
#elif 0
#  define ALGORITHM_BYTE_ORDER IS_BIG_ENDIAN
#else
#  error The algorithm byte order is not defined
#endif

/*  2. VIA ACE SUPPORT */

#if defined( __GNUC__ ) && defined( __i386__ ) \
 || defined( _WIN32   ) && defined( _M_IX86  ) \
 && !(defined( _WIN64 ) || defined( _WIN32_WCE ) || defined( _MSC_VER
↪ ) && ( _MSC_VER <= 800 ))
#  define VIA_ACE_POSSIBLE
#endif

/*  Define this option if support for the VIA ACE is required. This
↪uses
    inline assembler instructions and is only implemented for the
↪Microsoft,
    Intel and GCC compilers.  If VIA ACE is known to be present, then
↪ defining
    ASSUME_VIA_ACE_PRESENT will remove the ordinary encryption/
↪decryption
    code.  If USE_VIA_ACE_IF_PRESENT is defined then VIA ACE will be
↪used if
    it is detected (both present and enabled) but the normal AES code
↪ will
    also be present.
```

```

    When VIA ACE is to be used, all AES encryption contexts MUST be 16 byte
    aligned; other input/output buffers do not need to be 16 byte aligned
    but there are very large performance gains if this can be arranged.
    VIA ACE also requires the decryption key schedule to be in reverse
    order (which later checks below ensure).
*/

#if 1 && defined( VIA_ACE_POSSIBLE ) && !defined( USE_VIA_ACE_IF_PRESENT )
#  define USE_VIA_ACE_IF_PRESENT
#endif

#if 0 && defined( VIA_ACE_POSSIBLE ) && !defined( ASSUME_VIA_ACE_PRESENT )
#  define ASSUME_VIA_ACE_PRESENT
#  endif

/*  3. ASSEMBLER SUPPORT

    This define (which can be on the command line) enables the use of the
    assembler code routines for encryption, decryption and key scheduling
    as follows:

    ASM_X86_V1C uses the assembler (aes_x86_v1.asm) with large tables for
                encryption and decryption and but with key scheduling in C
    ASM_X86_V2  uses assembler (aes_x86_v2.asm) with compressed tables for
                encryption, decryption and key scheduling
    ASM_X86_V2C uses assembler (aes_x86_v2.asm) with compressed tables for
                encryption and decryption and but with key scheduling in C
    ASM_AMD64_C uses assembler (aes_amd64.asm) with compressed tables for
                encryption and decryption and but with key scheduling in C

    Change one 'if 0' below to 'if 1' to select the version or define
    as a compilation option.
*/
```

```

#if 0 && !defined( ASM_X86_V1C )
#  define ASM_X86_V1C
#elif 0 && !defined( ASM_X86_V2  )
#  define ASM_X86_V2
#elif 0 && !defined( ASM_X86_V2C )
#  define ASM_X86_V2C
#elif 0 && !defined( ASM_AMD64_C )
#  define ASM_AMD64_C
#endif

#if (defined ( ASM_X86_V1C ) || defined( ASM_X86_V2 ) || defined( ASM_X86_V2C )) \
      && !defined( _M_IX86 ) || defined( ASM_AMD64_C ) && !defined( _M_X64 )
#  error Assembler code is only available for x86 and AMD64 systems
#endif

/*  4. FAST INPUT/OUTPUT OPERATIONS.

    On some machines it is possible to improve speed by transferring the
    bytes in the input and output arrays to and from the internal 32-bit
    variables by addressing these arrays as if they are arrays of 32-bit
    words.  On some machines this will always be possible but there may
    be a large performance penalty if the byte arrays are not aligned on
    the normal word boundaries. On other machines this technique will
    lead to memory access errors when such 32-bit word accesses are not
    properly aligned. The option SAFE_IO avoids such problems but will
    often be slower on those machines that support misaligned access
    (especially so if care is taken to align the input  and output byte
    arrays on 32-bit word boundaries). If SAFE_IO is not defined it is
    assumed that access to byte arrays as if they are arrays of 32-bit
    words will not cause problems when such accesses are misaligned.
*/
#if 1 && !defined( _MSC_VER )
#  define SAFE_IO
#endif

/*  5. LOOP UNROLLING
```

```

    The code for encryption and decrytpion cycles through a number of rounds
    that can be implemented either in a loop or by expanding the code into a
    long sequence of instructions, the latter producing a larger program but
    one that will often be much faster. The latter is called loop unrolling.
    There are also potential speed advantages in expanding two iterations in
    a loop with half the number of iterations, which is called partial loop
    unrolling.  The following options allow partial or full loop unrolling
    to be set independently for encryption and decryption
*/
#if 1
#  define ENC_UNROLL  FULL
#elif 0
#  define ENC_UNROLL  PARTIAL
#else
#  define ENC_UNROLL  NONE
#endif

#if 1
#  define DEC_UNROLL  FULL
#elif 0
#  define DEC_UNROLL  PARTIAL
#else
#  define DEC_UNROLL  NONE
#endif

#if 1
#  define ENC_KS_UNROLL
#endif

#if 1
#  define DEC_KS_UNROLL
#endif

/*  6. FAST FINITE FIELD OPERATIONS

    If this section is included, tables are used to provide faster finite
    field arithmetic (this has no effect if FIXED_TABLES is defined).
*/
#if 1
#  define FF_TABLES
```

```
#endif

/*  7. INTERNAL STATE VARIABLE FORMAT

    The internal state of Rijndael is stored in a number of local 32-bit
    word varaibles which can be defined either as an array or as individual
    names variables. Include this section if you want to store these local
    varaibles in arrays. Otherwise individual local variables will be used.
*/
#if 1
#  define ARRAYS
#endif

/*  8. FIXED OR DYNAMIC TABLES

    When this section is included the tables used by the code are compiled
    statically into the binary file.  Otherwise the subroutine aes_init()
    must be called to compute them before the code is first used.
*/
#if 1 && !(defined( _MSC_VER ) && ( _MSC_VER <= 800 ))
#  define FIXED_TABLES
#endif

/*  9. MASKING OR CASTING FROM LONGER VALUES TO BYTES

    In some systems it is better to mask longer values to extract bytes
    rather than using a cast. This option allows this choice.
*/
#if 0
#  define to_byte(x)  ((uint_8t)(x))
#else
#  define to_byte(x)  ((x) & 0xff)
#endif

/*  10. TABLE ALIGNMENT

    On some sytsems speed will be improved by aligning the AES large lookup
    tables on particular boundaries. This define should be set to a power of
    two giving the desired alignment. It can be left undefined if alignment
```

```
    is not needed.  This option is specific to the Microsft VC++
↪compiler -
    it seems to sometimes cause trouble for the VC++ version 6
↪compiler.
*/

#if 1 && defined( _MSC_VER ) && ( _MSC_VER >= 1300 )
#  define TABLE_ALIGN 32
#endif

/*  11.  REDUCE CODE AND TABLE SIZE

    This replaces some expanded macros with function calls if
↪AES_ASM_V2 or
    AES_ASM_V2C are defined
*/

#if 1 && (defined( ASM_X86_V2 ) || defined( ASM_X86_V2C ))
#  define REDUCE_CODE_SIZE
#endif

/*  12. TABLE OPTIONS

    This cipher proceeds by repeating in a number of cycles known as
↪'rounds'
    which are implemented by a round function which can optionally be
↪ speeded
    up using tables.  The basic tables are each 256 32-bit words,
↪with either
    one or four tables being required for each round function
↪depending on
    how much speed is required. The encryption and decryption round
↪functions
    are different and the last encryption and decrytpion round
↪functions are
    different again making four different round functions in all.

    This means that:
      1. Normal encryption and decryption rounds can each use either
↪0, 1
         or 4 tables and table spaces of 0, 1024 or 4096 bytes each.
      2. The last encryption and decryption rounds can also use
↪either 0, 1
         or 4 tables and table spaces of 0, 1024 or 4096 bytes each.

    Include or exclude the appropriate definitions below to set the
↪number
    of tables used by this implementation.
*/
```

```
#if 1   /* set tables for the normal encryption round */
#   define ENC_ROUND   FOUR_TABLES
#elif 0
#   define ENC_ROUND   ONE_TABLE
#else
#   define ENC_ROUND   NO_TABLES
#endif

#if 1   /* set tables for the last encryption round */
#   define LAST_ENC_ROUND  FOUR_TABLES
#elif 0
#   define LAST_ENC_ROUND  ONE_TABLE
#else
#   define LAST_ENC_ROUND  NO_TABLES
#endif

#if 1   /* set tables for the normal decryption round */
#   define DEC_ROUND   FOUR_TABLES
#elif 0
#   define DEC_ROUND   ONE_TABLE
#else
#   define DEC_ROUND   NO_TABLES
#endif

#if 1   /* set tables for the last decryption round */
#   define LAST_DEC_ROUND  FOUR_TABLES
#elif 0
#   define LAST_DEC_ROUND  ONE_TABLE
#else
#   define LAST_DEC_ROUND  NO_TABLES
#endif

/*  The decryption key schedule can be speeded up with tables in the
↪same
    way that the round functions can.  Include or exclude the
↪following
    defines to set this requirement.
*/
#if 1
#   define KEY_SCHED   FOUR_TABLES
#elif 0
#   define KEY_SCHED   ONE_TABLE
#else
#   define KEY_SCHED   NO_TABLES
#endif

/*  ---- END OF USER CONFIGURED OPTIONS ---- */

```

```
/* VIA ACE support is only available for VC++ and GCC */

#if !defined( _MSC_VER ) && !defined( __GNUC__ )
#  if defined( ASSUME_VIA_ACE_PRESENT )
#    undef ASSUME_VIA_ACE_PRESENT
#  endif
#  if defined( USE_VIA_ACE_IF_PRESENT )
#    undef USE_VIA_ACE_IF_PRESENT
#  endif
#endif

#if defined( ASSUME_VIA_ACE_PRESENT ) && !defined( USE_VIA_ACE_IF_PRESENT )
#  define USE_VIA_ACE_IF_PRESENT
#endif

#if defined( USE_VIA_ACE_IF_PRESENT ) && !defined ( AES_REV_DKS )
#  define AES_REV_DKS
#endif

/* Assembler support requires the use of platform byte order */

#if ( defined( ASM_X86_V1C ) || defined( ASM_X86_V2C ) || defined( ASM_AMD64_C ) ) \
    && (ALGORITHM_BYTE_ORDER != PLATFORM_BYTE_ORDER)
#  undef  ALGORITHM_BYTE_ORDER
#  define ALGORITHM_BYTE_ORDER PLATFORM_BYTE_ORDER
#endif

/* In this implementation the columns of the state array are each held in
   32-bit words. The state array can be held in various ways: in an array
   of words, in a number of individual word variables or in a number of
   processor registers. The following define maps a variable name x and
   a column number c to the way the state array variable is to be held.
   The first define below maps the state into an array x[c] whereas the
   second form maps the state into a number of individual variables x0,
   x1, etc.  Another form could map individual state colums to machine
   register names.
*/

#if defined( ARRAYS )
```

```
#  define s(x,c) x[c]
#else
#  define s(x,c) x##c
#endif

/*  This implementation provides subroutines for encryption, decryption
    and for setting the three key lengths (separately) for encryption
    and decryption. Since not all functions are needed, masks are set
    up here to determine which will be implemented in C
*/

#if !defined( AES_ENCRYPT )
#  define EFUNCS_IN_C   0
#elif defined( ASSUME_VIA_ACE_PRESENT ) || defined( ASM_X86_V1C ) \
    || defined( ASM_X86_V2C ) || defined( ASM_AMD64_C )
#  define EFUNCS_IN_C   ENC_KEYING_IN_C
#elif !defined( ASM_X86_V2 )
#  define EFUNCS_IN_C   ( ENCRYPTION_IN_C | ENC_KEYING_IN_C )
#else
#  define EFUNCS_IN_C   0
#endif

#if !defined( AES_DECRYPT )
#  define DFUNCS_IN_C   0
#elif defined( ASSUME_VIA_ACE_PRESENT ) || defined( ASM_X86_V1C ) \
    || defined( ASM_X86_V2C ) || defined( ASM_AMD64_C )
#  define DFUNCS_IN_C   DEC_KEYING_IN_C
#elif !defined( ASM_X86_V2 )
#  define DFUNCS_IN_C   ( DECRYPTION_IN_C | DEC_KEYING_IN_C )
#else
#  define DFUNCS_IN_C   0
#endif

#define FUNCS_IN_C  ( EFUNCS_IN_C | DFUNCS_IN_C )

/* END OF CONFIGURATION OPTIONS */

#define RC_LENGTH   (5 * (AES_BLOCK_SIZE / 4 - 2))

/* Disable or report errors on some combinations of options */

#if ENC_ROUND == NO_TABLES && LAST_ENC_ROUND != NO_TABLES
#  undef  LAST_ENC_ROUND
#  define LAST_ENC_ROUND  NO_TABLES
#elif ENC_ROUND == ONE_TABLE && LAST_ENC_ROUND == FOUR_TABLES
#  undef  LAST_ENC_ROUND
#  define LAST_ENC_ROUND  ONE_TABLE
#endif
```

```

#if ENC_ROUND == NO_TABLES && ENC_UNROLL != NONE
#  undef  ENC_UNROLL
#  define ENC_UNROLL  NONE
#endif

#if DEC_ROUND == NO_TABLES && LAST_DEC_ROUND != NO_TABLES
#  undef  LAST_DEC_ROUND
#  define LAST_DEC_ROUND  NO_TABLES
#elif DEC_ROUND == ONE_TABLE && LAST_DEC_ROUND == FOUR_TABLES
#  undef  LAST_DEC_ROUND
#  define LAST_DEC_ROUND  ONE_TABLE
#endif

#if DEC_ROUND == NO_TABLES && DEC_UNROLL != NONE
#  undef  DEC_UNROLL
#  define DEC_UNROLL  NONE
#endif

#if defined( bswap32 )
#  define aes_sw32    bswap32
#elif defined( bswap_32 )
#  define aes_sw32    bswap_32
#else
#  define brot(x,n)   (((uint_32t)(x) <<  n) | ((uint_32t)(x) >> (32 - n)))
#  define aes_sw32(x) ((brot((x),8) & 0x00ff00ff) | (brot((x),24) & 0xff00ff00))
#endif

/*  upr(x,n):  rotates bytes within words by n positions, moving bytes to
               higher index positions with wrap around into low positions
    ups(x,n):  moves bytes by n positions to higher index positions in
               words but without wrap around
    bval(x,n): extracts a byte from a word

    WARNING:   The definitions given here are intended only for use with
               unsigned variables and with shift counts that are compile
               time constants
*/

#if ( ALGORITHM_BYTE_ORDER == IS_LITTLE_ENDIAN )
#  define upr(x,n)      (((uint_32t)(x) << (8 * (n))) | ((uint_32t)(x) >> (32 - 8 * (n))))
```

```
#  define ups(x,n)      ((uint_32t) (x) << (8 * (n)))
#  define bval(x,n)     to_byte((x) >> (8 * (n)))
#  define bytes2word(b0, b1, b2, b3)  \
        (((uint_32t)(b3) << 24) | ((uint_32t)(b2) << 16) | ((uint_32t
↪)(b1) << 8) | (b0))
#endif

#if ( ALGORITHM_BYTE_ORDER == IS_BIG_ENDIAN )
#  define upr(x,n)      (((uint_32t)(x) >> (8 * (n))) | ((uint_32t)(x
↪) << (32 - 8 * (n))))
#  define ups(x,n)      ((uint_32t) (x) >> (8 * (n)))
#  define bval(x,n)     to_byte((x) >> (24 - 8 * (n)))
#  define bytes2word(b0, b1, b2, b3)  \
        (((uint_32t)(b0) << 24) | ((uint_32t)(b1) << 16) | ((uint_32t
↪)(b2) << 8) | (b3))
#endif

#if defined( SAFE_IO )
#  define word_in(x,c)    bytes2word(((const uint_8t*)(x)+4*c)[0], ((
↪const uint_8t*)(x)+4*c)[1], \
                                   ((const uint_8t*)(x)+4*c)[2], ((
↪const uint_8t*)(x)+4*c)[3])
#  define word_out(x,c,v) { ((uint_8t*)(x)+4*c)[0] = bval(v,0); ((
↪uint_8t*)(x)+4*c)[1] = bval(v,1); \
                          ((uint_8t*)(x)+4*c)[2] = bval(v,2); ((
↪uint_8t*)(x)+4*c)[3] = bval(v,3); }
#elif ( ALGORITHM_BYTE_ORDER == PLATFORM_BYTE_ORDER )
#  define word_in(x,c)    (*((uint_32t*)(x)+(c)))
#  define word_out(x,c,v) (*((uint_32t*)(x)+(c)) = (v))
#else
#  define word_in(x,c)    aes_sw32(*((uint_32t*)(x)+(c)))
#  define word_out(x,c,v) (*((uint_32t*)(x)+(c)) = aes_sw32(v))
#endif

/* the finite field modular polynomial and elements */

#define WPOLY   0x011b
#define BPOLY     0x1b

/* multiply four bytes in GF(2^8) by 'x' {02} in parallel */

#define gf_c1  0x80808080
#define gf_c2  0x7f7f7f7f
#define gf_mulx(x)  ((((x) & gf_c2) << 1) ^ ((((x) & gf_c1) >> 7) *
↪BPOLY))

/* The following defines provide alternative definitions of gf_mulx
↪that might
   give improved performance if a fast 32-bit multiply is not
```

```
↪available. Note
    that a temporary variable u needs to be defined where gf_mulx is
↪used.

#define gf_mulx(x) (u = (x) & gf_c1, u |= (u >> 1), ((x) & gf_c2) <<
↪1) ^ ((u >> 3) | (u >> 6))
#define gf_c4  (0x01010101 * BPOLY)
#define gf_mulx(x) (u = (x) & gf_c1, ((x) & gf_c2) << 1) ^ ((u - (u
↪>> 7)) & gf_c4)
*/

/* Work out which tables are needed for the different options   */

#if defined( ASM_X86_V1C )
#   if defined( ENC_ROUND )
#     undef  ENC_ROUND
#   endif
#   define ENC_ROUND   FOUR_TABLES
#   if defined( LAST_ENC_ROUND )
#     undef  LAST_ENC_ROUND
#   endif
#   define LAST_ENC_ROUND  FOUR_TABLES
#   if defined( DEC_ROUND )
#     undef  DEC_ROUND
#   endif
#   define DEC_ROUND   FOUR_TABLES
#   if defined( LAST_DEC_ROUND )
#     undef  LAST_DEC_ROUND
#   endif
#   define LAST_DEC_ROUND  FOUR_TABLES
#   if defined( KEY_SCHED )
#     undef  KEY_SCHED
#     define KEY_SCHED   FOUR_TABLES
#   endif
#endif

#if ( FUNCS_IN_C & ENCRYPTION_IN_C ) || defined( ASM_X86_V1C )
#   if ENC_ROUND == ONE_TABLE
#     define FT1_SET
#   elif ENC_ROUND == FOUR_TABLES
#     define FT4_SET
#   else
#     define SBX_SET
#   endif
#   if LAST_ENC_ROUND == ONE_TABLE
#     define FL1_SET
#   elif LAST_ENC_ROUND == FOUR_TABLES
#     define FL4_SET
#   elif !defined( SBX_SET )
```

```c
#    define SBX_SET
#  endif
#endif

#if ( FUNCS_IN_C & DECRYPTION_IN_C ) || defined( ASM_X86_V1C )
#  if DEC_ROUND == ONE_TABLE
#    define IT1_SET
#  elif DEC_ROUND == FOUR_TABLES
#    define IT4_SET
#  else
#    define ISB_SET
#  endif
#  if LAST_DEC_ROUND == ONE_TABLE
#    define IL1_SET
#  elif LAST_DEC_ROUND == FOUR_TABLES
#    define IL4_SET
#  elif !defined(ISB_SET)
#    define ISB_SET
#  endif
#endif

#if !(defined( REDUCE_CODE_SIZE ) && (defined( ASM_X86_V2 ) || defined( ASM_X86_V2C )))
#  if ((FUNCS_IN_C & ENC_KEYING_IN_C) || (FUNCS_IN_C & DEC_KEYING_IN_C))
#    if KEY_SCHED == ONE_TABLE
#      if !defined( FL1_SET )  && !defined( FL4_SET )
#        define LS1_SET
#      endif
#    elif KEY_SCHED == FOUR_TABLES
#      if !defined( FL4_SET )
#        define LS4_SET
#      endif
#    elif !defined( SBX_SET )
#      define SBX_SET
#    endif
#  endif
#  if (FUNCS_IN_C & DEC_KEYING_IN_C)
#    if KEY_SCHED == ONE_TABLE
#      define IM1_SET
#    elif KEY_SCHED == FOUR_TABLES
#      define IM4_SET
#    elif !defined( SBX_SET )
#      define SBX_SET
#    endif
#  endif
#endif

/* generic definitions of Rijndael macros that use tables    */
```

```
#define no_table(x,box,vf,rf,c) bytes2word( \
    box[bval(vf(x,0,c),rf(0,c))], \
    box[bval(vf(x,1,c),rf(1,c))], \
    box[bval(vf(x,2,c),rf(2,c))], \
    box[bval(vf(x,3,c),rf(3,c))])

#define one_table(x,op,tab,vf,rf,c) \
 (     tab[bval(vf(x,0,c),rf(0,c))] \
  ^ op(tab[bval(vf(x,1,c),rf(1,c))],1) \
  ^ op(tab[bval(vf(x,2,c),rf(2,c))],2) \
  ^ op(tab[bval(vf(x,3,c),rf(3,c))],3))

#define four_tables(x,tab,vf,rf,c) \
 (  tab[0][bval(vf(x,0,c),rf(0,c))] \
  ^ tab[1][bval(vf(x,1,c),rf(1,c))] \
  ^ tab[2][bval(vf(x,2,c),rf(2,c))] \
  ^ tab[3][bval(vf(x,3,c),rf(3,c))])

#define vf1(x,r,c)  (x)
#define rf1(r,c)    (r)
#define rf2(r,c)    ((8+r-c)&3)

/* perform forward and inverse column mix operation on four bytes in
↪long word x in */
/* parallel. NOTE: x must be a simple variable, NOT an expression in
↪these macros.  */

#if !(defined( REDUCE_CODE_SIZE ) && (defined( ASM_X86_V2 ) ||
↪defined( ASM_X86_V2C )))

#if defined( FM4_SET )       /* not currently used */
#  define fwd_mcol(x)        four_tables(x,t_use(f,m),vf1,rf1,0)
#elif defined( FM1_SET )     /* not currently used */
#  define fwd_mcol(x)        one_table(x,upr,t_use(f,m),vf1,rf1,0)
#else
#  define dec_fmvars         uint_32t g2
#  define fwd_mcol(x)        (g2 = gf_mulx(x), g2 ^ upr((x) ^ g2, 3) ^
↪ upr((x), 2) ^ upr((x), 1))
#endif

#if defined( IM4_SET )
#  define inv_mcol(x)        four_tables(x,t_use(i,m),vf1,rf1,0)
#elif defined( IM1_SET )
#  define inv_mcol(x)        one_table(x,upr,t_use(i,m),vf1,rf1,0)
#else
#  define dec_imvars         uint_32t g2, g4, g9
#  define inv_mcol(x)        (g2 = gf_mulx(x), g4 = gf_mulx(g2), g9 =
↪(x) ^ gf_mulx(g4), g4 ^= g9, \
```

```
                             (x) ^ g2 ^ g4 ^ upr(g2 ^ g9, 3) ^ upr(g4,
↪ 2) ^ upr(g9, 1))
#endif

#if defined( FL4_SET )
#   define ls_box(x,c)       four_tables(x,t_use(f,l),vf1,rf2,c)
#elif defined( LS4_SET )
#   define ls_box(x,c)       four_tables(x,t_use(l,s),vf1,rf2,c)
#elif defined( FL1_SET )
#   define ls_box(x,c)       one_table(x,upr,t_use(f,l),vf1,rf2,c)
#elif defined( LS1_SET )
#   define ls_box(x,c)       one_table(x,upr,t_use(l,s),vf1,rf2,c)
#else
#   define ls_box(x,c)       no_table(x,t_use(s,box),vf1,rf2,c)
#endif

#endif

#if defined( ASM_X86_V1C ) && defined( AES_DECRYPT ) && !defined(
↪ISB_SET )
#   define ISB_SET
#endif

#endif
```

src/Obf/aestab.c

```
/*
---------------------------------------------------------------------------
↪
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights
↪reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied
↪warranties
in respect of its operation, including, but not limited to,
↪correctness
and fitness for purpose.
---------------------------------------------------------------------------
↪
Issue Date: 20/12/2007
```

```c
*/

#define DO_TABLES

#include "aes.h"
#include "aesopt.h"

#if defined(FIXED_TABLES)

#define sb_data(w) {\
    w(0x63), w(0x7c), w(0x77), w(0x7b), w(0xf2), w(0x6b), w(0x6f), w(0xc5),\
    w(0x30), w(0x01), w(0x67), w(0x2b), w(0xfe), w(0xd7), w(0xab), w(0x76),\
    w(0xca), w(0x82), w(0xc9), w(0x7d), w(0xfa), w(0x59), w(0x47), w(0xf0),\
    w(0xad), w(0xd4), w(0xa2), w(0xaf), w(0x9c), w(0xa4), w(0x72), w(0xc0),\
    w(0xb7), w(0xfd), w(0x93), w(0x26), w(0x36), w(0x3f), w(0xf7), w(0xcc),\
    w(0x34), w(0xa5), w(0xe5), w(0xf1), w(0x71), w(0xd8), w(0x31), w(0x15),\
    w(0x04), w(0xc7), w(0x23), w(0xc3), w(0x18), w(0x96), w(0x05), w(0x9a),\
    w(0x07), w(0x12), w(0x80), w(0xe2), w(0xeb), w(0x27), w(0xb2), w(0x75),\
    w(0x09), w(0x83), w(0x2c), w(0x1a), w(0x1b), w(0x6e), w(0x5a), w(0xa0),\
    w(0x52), w(0x3b), w(0xd6), w(0xb3), w(0x29), w(0xe3), w(0x2f), w(0x84),\
    w(0x53), w(0xd1), w(0x00), w(0xed), w(0x20), w(0xfc), w(0xb1), w(0x5b),\
    w(0x6a), w(0xcb), w(0xbe), w(0x39), w(0x4a), w(0x4c), w(0x58), w(0xcf),\
    w(0xd0), w(0xef), w(0xaa), w(0xfb), w(0x43), w(0x4d), w(0x33), w(0x85),\
    w(0x45), w(0xf9), w(0x02), w(0x7f), w(0x50), w(0x3c), w(0x9f), w(0xa8),\
    w(0x51), w(0xa3), w(0x40), w(0x8f), w(0x92), w(0x9d), w(0x38), w(0xf5),\
    w(0xbc), w(0xb6), w(0xda), w(0x21), w(0x10), w(0xff), w(0xf3), w(0xd2),\
    w(0xcd), w(0x0c), w(0x13), w(0xec), w(0x5f), w(0x97), w(0x44), w(0x17),\
    w(0xc4), w(0xa7), w(0x7e), w(0x3d), w(0x64), w(0x5d), w(0x19), w(0x73),\
    w(0x60), w(0x81), w(0x4f), w(0xdc), w(0x22), w(0x2a), w(0x90), w(0x88),\
    w(0x46), w(0xee), w(0xb8), w(0x14), w(0xde), w(0x5e), w(0x0b), w
```

```
↪(0xdb),\
    w(0xe0), w(0x32), w(0x3a), w(0x0a), w(0x49), w(0x06), w(0x24), w
↪(0x5c),\
    w(0xc2), w(0xd3), w(0xac), w(0x62), w(0x91), w(0x95), w(0xe4), w
↪(0x79),\
    w(0xe7), w(0xc8), w(0x37), w(0x6d), w(0x8d), w(0xd5), w(0x4e), w
↪(0xa9),\
    w(0x6c), w(0x56), w(0xf4), w(0xea), w(0x65), w(0x7a), w(0xae), w
↪(0x08),\
    w(0xba), w(0x78), w(0x25), w(0x2e), w(0x1c), w(0xa6), w(0xb4), w
↪(0xc6),\
    w(0xe8), w(0xdd), w(0x74), w(0x1f), w(0x4b), w(0xbd), w(0x8b), w
↪(0x8a),\
    w(0x70), w(0x3e), w(0xb5), w(0x66), w(0x48), w(0x03), w(0xf6), w
↪(0x0e),\
    w(0x61), w(0x35), w(0x57), w(0xb9), w(0x86), w(0xc1), w(0x1d), w
↪(0x9e),\
    w(0xe1), w(0xf8), w(0x98), w(0x11), w(0x69), w(0xd9), w(0x8e), w
↪(0x94),\
    w(0x9b), w(0x1e), w(0x87), w(0xe9), w(0xce), w(0x55), w(0x28), w
↪(0xdf),\
    w(0x8c), w(0xa1), w(0x89), w(0x0d), w(0xbf), w(0xe6), w(0x42), w
↪(0x68),\
    w(0x41), w(0x99), w(0x2d), w(0x0f), w(0xb0), w(0x54), w(0xbb), w
↪(0x16) }

#define isb_data(w) {\
    w(0x52), w(0x09), w(0x6a), w(0xd5), w(0x30), w(0x36), w(0xa5), w
↪(0x38),\
    w(0xbf), w(0x40), w(0xa3), w(0x9e), w(0x81), w(0xf3), w(0xd7), w
↪(0xfb),\
    w(0x7c), w(0xe3), w(0x39), w(0x82), w(0x9b), w(0x2f), w(0xff), w
↪(0x87),\
    w(0x34), w(0x8e), w(0x43), w(0x44), w(0xc4), w(0xde), w(0xe9), w
↪(0xcb),\
    w(0x54), w(0x7b), w(0x94), w(0x32), w(0xa6), w(0xc2), w(0x23), w
↪(0x3d),\
    w(0xee), w(0x4c), w(0x95), w(0x0b), w(0x42), w(0xfa), w(0xc3), w
↪(0x4e),\
    w(0x08), w(0x2e), w(0xa1), w(0x66), w(0x28), w(0xd9), w(0x24), w
↪(0xb2),\
    w(0x76), w(0x5b), w(0xa2), w(0x49), w(0x6d), w(0x8b), w(0xd1), w
↪(0x25),\
    w(0x72), w(0xf8), w(0xf6), w(0x64), w(0x86), w(0x68), w(0x98), w
↪(0x16),\
    w(0xd4), w(0xa4), w(0x5c), w(0xcc), w(0x5d), w(0x65), w(0xb6), w
↪(0x92),\
    w(0x6c), w(0x70), w(0x48), w(0x50), w(0xfd), w(0xed), w(0xb9), w
↪(0xda),\
```

```
    w(0x5e), w(0x15), w(0x46), w(0x57), w(0xa7), w(0x8d), w(0x9d), w
↪(0x84),\
    w(0x90), w(0xd8), w(0xab), w(0x00), w(0x8c), w(0xbc), w(0xd3), w
↪(0x0a),\
    w(0xf7), w(0xe4), w(0x58), w(0x05), w(0xb8), w(0xb3), w(0x45), w
↪(0x06),\
    w(0xd0), w(0x2c), w(0x1e), w(0x8f), w(0xca), w(0x3f), w(0x0f), w
↪(0x02),\
    w(0xc1), w(0xaf), w(0xbd), w(0x03), w(0x01), w(0x13), w(0x8a), w
↪(0x6b),\
    w(0x3a), w(0x91), w(0x11), w(0x41), w(0x4f), w(0x67), w(0xdc), w
↪(0xea),\
    w(0x97), w(0xf2), w(0xcf), w(0xce), w(0xf0), w(0xb4), w(0xe6), w
↪(0x73),\
    w(0x96), w(0xac), w(0x74), w(0x22), w(0xe7), w(0xad), w(0x35), w
↪(0x85),\
    w(0xe2), w(0xf9), w(0x37), w(0xe8), w(0x1c), w(0x75), w(0xdf), w
↪(0x6e),\
    w(0x47), w(0xf1), w(0x1a), w(0x71), w(0x1d), w(0x29), w(0xc5), w
↪(0x89),\
    w(0x6f), w(0xb7), w(0x62), w(0x0e), w(0xaa), w(0x18), w(0xbe), w
↪(0x1b),\
    w(0xfc), w(0x56), w(0x3e), w(0x4b), w(0xc6), w(0xd2), w(0x79), w
↪(0x20),\
    w(0x9a), w(0xdb), w(0xc0), w(0xfe), w(0x78), w(0xcd), w(0x5a), w
↪(0xf4),\
    w(0x1f), w(0xdd), w(0xa8), w(0x33), w(0x88), w(0x07), w(0xc7), w
↪(0x31),\
    w(0xb1), w(0x12), w(0x10), w(0x59), w(0x27), w(0x80), w(0xec), w
↪(0x5f),\
    w(0x60), w(0x51), w(0x7f), w(0xa9), w(0x19), w(0xb5), w(0x4a), w
↪(0x0d),\
    w(0x2d), w(0xe5), w(0x7a), w(0x9f), w(0x93), w(0xc9), w(0x9c), w
↪(0xef),\
    w(0xa0), w(0xe0), w(0x3b), w(0x4d), w(0xae), w(0x2a), w(0xf5), w
↪(0xb0),\
    w(0xc8), w(0xeb), w(0xbb), w(0x3c), w(0x83), w(0x53), w(0x99), w
↪(0x61),\
    w(0x17), w(0x2b), w(0x04), w(0x7e), w(0xba), w(0x77), w(0xd6), w
↪(0x26),\
    w(0xe1), w(0x69), w(0x14), w(0x63), w(0x55), w(0x21), w(0x0c), w
↪(0x7d) }

#define mm_data(w) {\
    w(0x00), w(0x01), w(0x02), w(0x03), w(0x04), w(0x05), w(0x06), w
↪(0x07),\
    w(0x08), w(0x09), w(0x0a), w(0x0b), w(0x0c), w(0x0d), w(0x0e), w
↪(0x0f),\
    w(0x10), w(0x11), w(0x12), w(0x13), w(0x14), w(0x15), w(0x16), w
```

```
↪(0x17),\
    w(0x18), w(0x19), w(0x1a), w(0x1b), w(0x1c), w(0x1d), w(0x1e), w
↪(0x1f),\
    w(0x20), w(0x21), w(0x22), w(0x23), w(0x24), w(0x25), w(0x26), w
↪(0x27),\
    w(0x28), w(0x29), w(0x2a), w(0x2b), w(0x2c), w(0x2d), w(0x2e), w
↪(0x2f),\
    w(0x30), w(0x31), w(0x32), w(0x33), w(0x34), w(0x35), w(0x36), w
↪(0x37),\
    w(0x38), w(0x39), w(0x3a), w(0x3b), w(0x3c), w(0x3d), w(0x3e), w
↪(0x3f),\
    w(0x40), w(0x41), w(0x42), w(0x43), w(0x44), w(0x45), w(0x46), w
↪(0x47),\
    w(0x48), w(0x49), w(0x4a), w(0x4b), w(0x4c), w(0x4d), w(0x4e), w
↪(0x4f),\
    w(0x50), w(0x51), w(0x52), w(0x53), w(0x54), w(0x55), w(0x56), w
↪(0x57),\
    w(0x58), w(0x59), w(0x5a), w(0x5b), w(0x5c), w(0x5d), w(0x5e), w
↪(0x5f),\
    w(0x60), w(0x61), w(0x62), w(0x63), w(0x64), w(0x65), w(0x66), w
↪(0x67),\
    w(0x68), w(0x69), w(0x6a), w(0x6b), w(0x6c), w(0x6d), w(0x6e), w
↪(0x6f),\
    w(0x70), w(0x71), w(0x72), w(0x73), w(0x74), w(0x75), w(0x76), w
↪(0x77),\
    w(0x78), w(0x79), w(0x7a), w(0x7b), w(0x7c), w(0x7d), w(0x7e), w
↪(0x7f),\
    w(0x80), w(0x81), w(0x82), w(0x83), w(0x84), w(0x85), w(0x86), w
↪(0x87),\
    w(0x88), w(0x89), w(0x8a), w(0x8b), w(0x8c), w(0x8d), w(0x8e), w
↪(0x8f),\
    w(0x90), w(0x91), w(0x92), w(0x93), w(0x94), w(0x95), w(0x96), w
↪(0x97),\
    w(0x98), w(0x99), w(0x9a), w(0x9b), w(0x9c), w(0x9d), w(0x9e), w
↪(0x9f),\
    w(0xa0), w(0xa1), w(0xa2), w(0xa3), w(0xa4), w(0xa5), w(0xa6), w
↪(0xa7),\
    w(0xa8), w(0xa9), w(0xaa), w(0xab), w(0xac), w(0xad), w(0xae), w
↪(0xaf),\
    w(0xb0), w(0xb1), w(0xb2), w(0xb3), w(0xb4), w(0xb5), w(0xb6), w
↪(0xb7),\
    w(0xb8), w(0xb9), w(0xba), w(0xbb), w(0xbc), w(0xbd), w(0xbe), w
↪(0xbf),\
    w(0xc0), w(0xc1), w(0xc2), w(0xc3), w(0xc4), w(0xc5), w(0xc6), w
↪(0xc7),\
    w(0xc8), w(0xc9), w(0xca), w(0xcb), w(0xcc), w(0xcd), w(0xce), w
↪(0xcf),\
    w(0xd0), w(0xd1), w(0xd2), w(0xd3), w(0xd4), w(0xd5), w(0xd6), w
↪(0xd7),\
```

```
    w(0xd8), w(0xd9), w(0xda), w(0xdb), w(0xdc), w(0xdd), w(0xde), w(0xdf),\
    w(0xe0), w(0xe1), w(0xe2), w(0xe3), w(0xe4), w(0xe5), w(0xe6), w(0xe7),\
    w(0xe8), w(0xe9), w(0xea), w(0xeb), w(0xec), w(0xed), w(0xee), w(0xef),\
    w(0xf0), w(0xf1), w(0xf2), w(0xf3), w(0xf4), w(0xf5), w(0xf6), w(0xf7),\
    w(0xf8), w(0xf9), w(0xfa), w(0xfb), w(0xfc), w(0xfd), w(0xfe), w(0xff) }

#define rc_data(w) {\
    w(0x01), w(0x02), w(0x04), w(0x08), w(0x10),w(0x20), w(0x40), w(0x80),\
    w(0x1b), w(0x36) }

#define h0(x)   (x)

#define w0(p)   bytes2word(p, 0, 0, 0)
#define w1(p)   bytes2word(0, p, 0, 0)
#define w2(p)   bytes2word(0, 0, p, 0)
#define w3(p)   bytes2word(0, 0, 0, p)

#define u0(p)   bytes2word(f2(p), p, p, f3(p))
#define u1(p)   bytes2word(f3(p), f2(p), p, p)
#define u2(p)   bytes2word(p, f3(p), f2(p), p)
#define u3(p)   bytes2word(p, p, f3(p), f2(p))

#define v0(p)   bytes2word(fe(p), f9(p), fd(p), fb(p))
#define v1(p)   bytes2word(fb(p), fe(p), f9(p), fd(p))
#define v2(p)   bytes2word(fd(p), fb(p), fe(p), f9(p))
#define v3(p)   bytes2word(f9(p), fd(p), fb(p), fe(p))

#endif

#if defined(FIXED_TABLES) || !defined(FF_TABLES)

#define f2(x)   ((x<<1) ^ (((x>>7) & 1) * WPOLY))
#define f4(x)   ((x<<2) ^ (((x>>6) & 1) * WPOLY) ^ (((x>>6) & 2) * WPOLY))
#define f8(x)   ((x<<3) ^ (((x>>5) & 1) * WPOLY) ^ (((x>>5) & 2) * WPOLY) \
                        ^ (((x>>5) & 4) * WPOLY))
#define f3(x)   (f2(x) ^ x)
#define f9(x)   (f8(x) ^ x)
#define fb(x)   (f8(x) ^ f2(x) ^ x)
#define fd(x)   (f8(x) ^ f4(x) ^ x)
#define fe(x)   (f8(x) ^ f4(x) ^ f2(x))

```

```c
#else

#define f2(x) ((x) ? pow[log[x] + 0x19] : 0)
#define f3(x) ((x) ? pow[log[x] + 0x01] : 0)
#define f9(x) ((x) ? pow[log[x] + 0xc7] : 0)
#define fb(x) ((x) ? pow[log[x] + 0x68] : 0)
#define fd(x) ((x) ? pow[log[x] + 0xee] : 0)
#define fe(x) ((x) ? pow[log[x] + 0xdf] : 0)

#endif

#include "aestab.h"

#if defined(__cplusplus)
extern "C"
{
#endif

#if defined(FIXED_TABLES)

/* implemented in case of wrong call for fixed tables */

AES_RETURN aes_init(void)
{
    return EXIT_SUCCESS;
}

#else   /*  Generate the tables for the dynamic table option */

#if defined(FF_TABLES)

#define gf_inv(x)   ((x) ? pow[ 255 - log[x]] : 0)

#else

/*  It will generally be sensible to use tables to compute finite
    field multiplies and inverses but where memory is scarse this
    code might sometimes be better. But it only has effect during
    initialisation so its pretty unimportant in overall terms.
*/

/*  return 2 ^ (n - 1) where n is the bit number of the highest bit
    set in x with x in the range 1 < x < 0x00000200.   This form is
    used so that locals within fi can be bytes rather than words
*/

static uint_8t hibit(const uint_32t x)
{   uint_8t r = (uint_8t)((x >> 1) | (x >> 2));

```

```
    r |= (r >> 2);
    r |= (r >> 4);
    return (r + 1) >> 1;
}

/* return the inverse of the finite field element x */

static uint_8t gf_inv(const uint_8t x)
{   uint_8t p1 = x, p2 = BPOLY, n1 = hibit(x), n2 = 0x80, v1 = 1, v2
↪= 0;

    if(x < 2)
        return x;

    for( ; ; )
    {
        if(n1)
            while(n2 >= n1)             /* divide polynomial p2 by p1
↪     */
            {
                n2 /= n1;               /* shift smaller polynomial
↪left */
                p2 ^= (p1 * n2) & 0xff; /* and remove from larger one
↪     */
                v2 ^= v1 * n2;          /* shift accumulated value
↪and    */
                n2 = hibit(p2);         /* add into result
↪               */
            }
        else
            return v1;

        if(n2)                          /* repeat with values swapped
↪     */
            while(n1 >= n2)
            {
                n1 /= n2;
                p1 ^= p2 * n1;
                v1 ^= v2 * n1;
                n1 = hibit(p1);
            }
        else
            return v2;
    }
}

#endif

/* The forward and inverse affine transformations used in the S-box
```

```
↪*/
uint_8t fwd_affine(const uint_8t x)
{   uint_32t w = x;
    w ^= (w << 1) ^ (w << 2) ^ (w << 3) ^ (w << 4);
    return 0x63 ^ ((w ^ (w >> 8)) & 0xff);
}

uint_8t inv_affine(const uint_8t x)
{   uint_32t w = x;
    w = (w << 1) ^ (w << 3) ^ (w << 6);
    return 0x05 ^ ((w ^ (w >> 8)) & 0xff);
}

static int init = 0;

AES_RETURN aes_init(void)
{   uint_32t  i, w;

#if defined(FF_TABLES)

    uint_8t  pow[512], log[256];

    if(init)
        return EXIT_SUCCESS;
    /*  log and power tables for GF(2^8) finite field with
        WPOLY as modular polynomial - the simplest primitive
        root is 0x03, used here to generate the tables
    */

    i = 0; w = 1;
    do
    {
        pow[i] = (uint_8t)w;
        pow[i + 255] = (uint_8t)w;
        log[w] = (uint_8t)i++;
        w ^=  (w << 1) ^ (w & 0x80 ? WPOLY : 0);
    }
    while (w != 1);

#else
    if(init)
        return EXIT_SUCCESS;
#endif

    for(i = 0, w = 1; i < RC_LENGTH; ++i)
    {
        t_set(r,c)[i] = bytes2word(w, 0, 0, 0);
        w = f2(w);
    }
```

```c

    for(i = 0; i < 256; ++i)
    {   uint_8t    b;

        b = fwd_affine(gf_inv((uint_8t)i));
        w = bytes2word(f2(b), b, b, f3(b));

#if defined( SBX_SET )
        t_set(s,box)[i] = b;
#endif

#if defined( FT1_SET )                 /* tables for a normal encryption round */
        t_set(f,n)[i] = w;
#endif
#if defined( FT4_SET )
        t_set(f,n)[0][i] = w;
        t_set(f,n)[1][i] = upr(w,1);
        t_set(f,n)[2][i] = upr(w,2);
        t_set(f,n)[3][i] = upr(w,3);
#endif
        w = bytes2word(b, 0, 0, 0);

#if defined( FL1_SET )            /* tables for last encryption round (may also   */
        t_set(f,l)[i] = w;        /* be used in the key schedule)                 */
#endif
#if defined( FL4_SET )
        t_set(f,l)[0][i] = w;
        t_set(f,l)[1][i] = upr(w,1);
        t_set(f,l)[2][i] = upr(w,2);
        t_set(f,l)[3][i] = upr(w,3);
#endif

#if defined( LS1_SET )                 /* table for key schedule if t_set(f,l) above is*/
        t_set(l,s)[i] = w;      /* not of the required form                     */
#endif
#if defined( LS4_SET )
        t_set(l,s)[0][i] = w;
        t_set(l,s)[1][i] = upr(w,1);
        t_set(l,s)[2][i] = upr(w,2);
        t_set(l,s)[3][i] = upr(w,3);
#endif

        b = gf_inv(inv_affine((uint_8t)i));
        w = bytes2word(fe(b), f9(b), fd(b), fb(b));
```

```

#if defined( IM1_SET )                   /* tables for the inverse mix
↪ column operation  */
        t_set(i,m)[b] = w;
#endif
#if defined( IM4_SET )
        t_set(i,m)[0][b] = w;
        t_set(i,m)[1][b] = upr(w,1);
        t_set(i,m)[2][b] = upr(w,2);
        t_set(i,m)[3][b] = upr(w,3);
#endif

#if defined( ISB_SET )
        t_set(i,box)[i] = b;
#endif
#if defined( IT1_SET )                   /* tables for a normal
↪decryption round */
        t_set(i,n)[i] = w;
#endif
#if defined( IT4_SET )
        t_set(i,n)[0][i] = w;
        t_set(i,n)[1][i] = upr(w,1);
        t_set(i,n)[2][i] = upr(w,2);
        t_set(i,n)[3][i] = upr(w,3);
#endif
        w = bytes2word(b, 0, 0, 0);
#if defined( IL1_SET )                   /* tables for last decryption
↪ round */
        t_set(i,l)[i] = w;
#endif
#if defined( IL4_SET )
        t_set(i,l)[0][i] = w;
        t_set(i,l)[1][i] = upr(w,1);
        t_set(i,l)[2][i] = upr(w,2);
        t_set(i,l)[3][i] = upr(w,3);
#endif
    }
    init = 1;
    return EXIT_SUCCESS;
}

#endif

#if defined(__cplusplus)
}
#endif
```

src/Obf/aestab.h

```
/*
```

```
---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 20/12/2007

 This file contains the code for declaring the tables needed to implement
 AES. The file aesopt.h is assumed to be included before this header file.
 If there are no global variables, the definitions here can be used to put
 the AES tables in a structure so that a pointer can then be added to the
 AES context to pass them to the AES routines that need them.   If this
 facility is used, the calling program has to ensure that this pointer is
 managed appropriately.  In particular, the value of the t_dec(in,it) item
 in the table structure must be set to zero in order to ensure that the
 tables are initialised. In practice the three code sequences in aeskey.c
 that control the calls to aes_init() and the aes_init() routine itself will
 have to be changed for a specific implementation. If global variables are
 available it will generally be preferable to use them with the precomputed
 FIXED_TABLES option that uses static global tables.

```

```
 The following defines can be used to control the way the tables
 are defined, initialised and used in embedded environments that
 require special features for these purposes

    the 't_dec' construction is used to declare fixed table arrays
    the 't_set' construction is used to set fixed table values
    the 't_use' construction is used to access fixed table values

    256 byte tables:

        t_xxx(s,box)    => forward S box
        t_xxx(i,box)    => inverse S box

    256 32-bit word OR 4 x 256 32-bit word tables:

        t_xxx(f,n)      => forward normal round
        t_xxx(f,l)      => forward last round
        t_xxx(i,n)      => inverse normal round
        t_xxx(i,l)      => inverse last round
        t_xxx(l,s)      => key schedule table
        t_xxx(i,m)      => key schedule table

    Other variables and tables:

        t_xxx(r,c)      => the rcon table
*/

#if !defined( _AESTAB_H )
#define _AESTAB_H

#if defined(__cplusplus)
extern "C" {
#endif

#define t_dec(m,n) t_##m##n
#define t_set(m,n) t_##m##n
#define t_use(m,n) t_##m##n

#if defined(FIXED_TABLES)
#  if !defined( __GNUC__ ) && (defined( __MSDOS__ ) || defined(
↪__WIN16__ ))
/*   make tables far data to avoid using too much DGROUP space (PG)
↪*/
#    define CONST const far
#  else
#    define CONST const
#  endif
#else
#  define CONST
```

```
#endif

#if defined(DO_TABLES)
#  define EXTERN
#else
#  define EXTERN extern
#endif

#if defined(_MSC_VER) && defined(TABLE_ALIGN)
#define ALIGN __declspec(align(TABLE_ALIGN))
#else
#define ALIGN
#endif

#if defined( __WATCOMC__ ) && ( __WATCOMC__ >= 1100 )
#  define XP_DIR __cdecl
#else
#  define XP_DIR
#endif

#if defined(DO_TABLES) && defined(FIXED_TABLES)
#define d_1(t,n,b,e)       EXTERN ALIGN CONST XP_DIR t n[256]    =
↪b(e)
#define d_4(t,n,b,e,f,g,h) EXTERN ALIGN CONST XP_DIR t n[4][256] = {
↪b(e), b(f), b(g), b(h) }
EXTERN ALIGN CONST uint_32t t_dec(r,c)[RC_LENGTH] = rc_data(w0);
#else
#define d_1(t,n,b,e)       EXTERN ALIGN CONST XP_DIR t n[256]
#define d_4(t,n,b,e,f,g,h) EXTERN ALIGN CONST XP_DIR t n[4][256]
EXTERN ALIGN CONST uint_32t t_dec(r,c)[RC_LENGTH];
#endif

#if defined( SBX_SET )
    d_1(uint_8t, t_dec(s,box), sb_data, h0);
#endif
#if defined( ISB_SET )
    d_1(uint_8t, t_dec(i,box), isb_data, h0);
#endif

#if defined( FT1_SET )
    d_1(uint_32t, t_dec(f,n), sb_data, u0);
#endif
#if defined( FT4_SET )
    d_4(uint_32t, t_dec(f,n), sb_data, u0, u1, u2, u3);
#endif

#if defined( FL1_SET )
    d_1(uint_32t, t_dec(f,l), sb_data, w0);
#endif
```

```
#if defined( FL4_SET )
    d_4(uint_32t, t_dec(f,l), sb_data, w0, w1, w2, w3);
#endif

#if defined( IT1_SET )
    d_1(uint_32t, t_dec(i,n), isb_data, v0);
#endif
#if defined( IT4_SET )
    d_4(uint_32t, t_dec(i,n), isb_data, v0, v1, v2, v3);
#endif

#if defined( IL1_SET )
    d_1(uint_32t, t_dec(i,l), isb_data, w0);
#endif
#if defined( IL4_SET )
    d_4(uint_32t, t_dec(i,l), isb_data, w0, w1, w2, w3);
#endif

#if defined( LS1_SET )
#if defined( FL1_SET )
#undef  LS1_SET
#else
    d_1(uint_32t, t_dec(l,s), sb_data, w0);
#endif
#endif

#if defined( LS4_SET )
#if defined( FL4_SET )
#undef  LS4_SET
#else
    d_4(uint_32t, t_dec(l,s), sb_data, w0, w1, w2, w3);
#endif
#endif

#if defined( IM1_SET )
    d_1(uint_32t, t_dec(i,m), mm_data, v0);
#endif
#if defined( IM4_SET )
    d_4(uint_32t, t_dec(i,m), mm_data, v0, v1, v2, v3);
#endif

#if defined(__cplusplus)
}
#endif

#endif
```

src/Obf/brg_endian.h

```
/*
```

```
---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 20/12/2007
*/

#ifndef _BRG_ENDIAN_H
#define _BRG_ENDIAN_H

#define IS_BIG_ENDIAN      4321 /* byte 0 is most significant (mc68k) */
#define IS_LITTLE_ENDIAN   1234 /* byte 0 is least significant (i386) */

/* Include files where endian defines and byteswap functions may reside */
#if defined( __sun )
#  include <sys/isa_defs.h>
#elif defined( __FreeBSD__ ) || defined( __OpenBSD__ ) || defined( __NetBSD__ )
#  include <sys/endian.h>
#elif defined( BSD ) && ( BSD >= 199103 ) || defined( __APPLE__ ) || \
      defined( __CYGWIN32__ ) || defined( __DJGPP__ ) || defined( __osf__ )
#  include <machine/endian.h>
#elif defined( __linux__ ) || defined( __GNUC__ ) || defined( __GNU_LIBRARY__ )
#  if !defined( __MINGW32__ ) && !defined( _AIX )
#    include <endian.h>
#    if !defined( __BEOS__ )
```

```c
#       include <byteswap.h>
#     endif
#   endif
#endif

/* Now attempt to set the define for platform byte order using any */
/* of the four forms SYMBOL, _SYMBOL, __SYMBOL & __SYMBOL__, which */
/* seem to encompass most endian symbol definitions */

#if defined( BIG_ENDIAN ) && defined( LITTLE_ENDIAN )
#   if defined( BYTE_ORDER ) && BYTE_ORDER == BIG_ENDIAN
#     define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#   elif defined( BYTE_ORDER ) && BYTE_ORDER == LITTLE_ENDIAN
#     define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#   endif
#elif defined( BIG_ENDIAN )
#   define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#elif defined( LITTLE_ENDIAN )
#   define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#endif

#if defined( _BIG_ENDIAN ) && defined( _LITTLE_ENDIAN )
#   if defined( _BYTE_ORDER ) && _BYTE_ORDER == _BIG_ENDIAN
#     define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#   elif defined( _BYTE_ORDER ) && _BYTE_ORDER == _LITTLE_ENDIAN
#     define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#   endif
#elif defined( _BIG_ENDIAN )
#   define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#elif defined( _LITTLE_ENDIAN )
#   define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#endif

#if defined( __BIG_ENDIAN ) && defined( __LITTLE_ENDIAN )
#   if defined( __BYTE_ORDER ) && __BYTE_ORDER == __BIG_ENDIAN
#     define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#   elif defined( __BYTE_ORDER ) && __BYTE_ORDER == __LITTLE_ENDIAN
#     define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#   endif
#elif defined( __BIG_ENDIAN )
#   define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#elif defined( __LITTLE_ENDIAN )
#   define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#endif

#if defined( __BIG_ENDIAN__ ) && defined( __LITTLE_ENDIAN__ )
```

```
#  if defined( __BYTE_ORDER__ ) && __BYTE_ORDER__ == __BIG_ENDIAN__
#    define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#  elif defined( __BYTE_ORDER__ ) && __BYTE_ORDER__ == __LITTLE_ENDIAN__
#    define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#  endif
#elif defined( __BIG_ENDIAN__ )
#  define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#elif defined( __LITTLE_ENDIAN__ )
#  define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#endif

/*  if the platform byte order could not be determined, then try to */
/*  set this define using common machine defines */
#if !defined(PLATFORM_BYTE_ORDER)

#if   defined( __alpha__ ) || defined( __alpha ) || defined( i386 )       || \
      defined( __i386__ )  || defined( _M_I86 )  || defined( _M_IX86 )    || \
      defined( __OS2__ )   || defined( sun386 )  || defined( __TURBOC__ ) || \
      defined( vax )       || defined( vms )     || defined( VMS )        || \
      defined( __VMS )     || defined( _M_X64 )
#  define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN

#elif defined( AMIGA )   || defined( applec )    || defined( __AS400__ )  || \
      defined( _CRAY )   || defined( __hppa )    || defined( __hp9000 )   || \
      defined( ibm370 )  || defined( mc68000 )   || defined( m68k )       || \
      defined( __MRC__ ) || defined( __MVS__ )   || defined( __MWERKS__ ) || \
      defined( sparc )   || defined( __sparc)    || defined( SYMANTEC_C ) || \
      defined( __VOS__ ) || defined( __TIGCC__ ) || defined( __TANDEM )   || \
      defined( THINK_C ) || defined( __VMCMS__ ) || defined( _AIX )
#  define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN

#elif 0     /* **** EDIT HERE IF NECESSARY **** */
#  define PLATFORM_BYTE_ORDER IS_LITTLE_ENDIAN
#elif 0     /* **** EDIT HERE IF NECESSARY **** */
#  define PLATFORM_BYTE_ORDER IS_BIG_ENDIAN
#else
```

```
#  error Please edit lines 126 or 128 in brg_endian.h to set the
↪platform byte order
#endif

#endif

#endif
```

src/Obf/brg_types.h

```
/*
---------------------------------------------------------------------------
↪
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights
↪reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied
↪warranties
in respect of its operation, including, but not limited to,
↪correctness
and fitness for purpose.
---------------------------------------------------------------------------
↪
Issue Date: 20/12/2007

 The unsigned integer types defined here are of the form uint_<nn>t
↪where
 <nn> is the length of the type; for example, the unsigned 32-bit
↪type is
 'uint_32t'.  These are NOT the same as the 'C99 integer types' that
↪are
 defined in the inttypes.h and stdint.h headers since attempts to use
↪ these
 types have shown that support for them is still highly variable.
↪However,
 since the latter are of the form uint<nn>_t, a regular expression
↪search
 and replace (in VC++ search on 'uint_{:z}t' and replace with 'uint\1
↪_t')
 can be used to convert the types used here to the C99 standard types
↪.
```

```
*/

#ifndef _BRG_TYPES_H
#define _BRG_TYPES_H

#if defined(__cplusplus)
extern "C" {
#endif

#include <limits.h>

#if defined( _MSC_VER ) && ( _MSC_VER >= 1300 )
#   include <stddef.h>
#   define ptrint_t intptr_t
#elif defined( __ECOS__ )
#   define intptr_t unsigned int
#   define ptrint_t intptr_t
#elif defined( __GNUC__ ) && ( __GNUC__ >= 3 )
#   include <stdint.h>
#   define ptrint_t intptr_t
#else
#   define ptrint_t int
#endif

#ifndef BRG_UI8
#   define BRG_UI8
#   if UCHAR_MAX == 255u
      typedef unsigned char uint_8t;
#   else
#     error Please define uint_8t as an 8-bit unsigned integer type in
↪ brg_types.h
#   endif
#endif

#ifndef BRG_UI16
#   define BRG_UI16
#   if USHRT_MAX == 65535u
      typedef unsigned short uint_16t;
#   else
#     error Please define uint_16t as a 16-bit unsigned short type in
↪brg_types.h
#   endif
#endif

#ifndef BRG_UI32
#   define BRG_UI32
#   if UINT_MAX == 4294967295u
#     define li_32(h) 0x##h##u
      typedef unsigned int uint_32t;
```

```
#   elif ULONG_MAX == 4294967295u
#     define li_32(h) 0x##h##ul
      typedef unsigned long uint_32t;
#   elif defined( _CRAY )
#     error This code needs 32-bit data types, which Cray machines do not provide
#   else
#     error Please define uint_32t as a 32-bit unsigned integer type in brg_types.h
#   endif
#endif

#ifndef BRG_UI64
#   if defined( __BORLANDC__ ) && !defined( __MSDOS__ )
#     define BRG_UI64
#     define li_64(h) 0x##h##ui64
      typedef unsigned __int64 uint_64t;
#   elif defined( _MSC_VER ) && ( _MSC_VER < 1300 )    /* 1300 == VC++ 7.0 */
#     define BRG_UI64
#     define li_64(h) 0x##h##ui64
      typedef unsigned __int64 uint_64t;
#   elif defined( __sun ) && defined( ULONG_MAX ) && ULONG_MAX == 0xfffffffful
#     define BRG_UI64
#     define li_64(h) 0x##h##ull
      typedef unsigned long long uint_64t;
#   elif defined( __MVS__ )
#     define BRG_UI64
#     define li_64(h) 0x##h##ull
      typedef unsigned int long long uint_64t;
#   elif defined( UINT_MAX ) && UINT_MAX > 4294967295u
#     if UINT_MAX == 18446744073709551615u
#       define BRG_UI64
#       define li_64(h) 0x##h##u
        typedef unsigned int uint_64t;
#     endif
#   elif defined( ULONG_MAX ) && ULONG_MAX > 4294967295u
#     if ULONG_MAX == 18446744073709551615ul
#       define BRG_UI64
#       define li_64(h) 0x##h##ul
        typedef unsigned long uint_64t;
#     endif
#   elif defined( ULLONG_MAX ) && ULLONG_MAX > 4294967295u
#     if ULLONG_MAX == 18446744073709551615ull
#       define BRG_UI64
#       define li_64(h) 0x##h##ull
        typedef unsigned long long uint_64t;
#     endif
```

```
#   elif defined( ULONG_LONG_MAX ) && ULONG_LONG_MAX > 4294967295u
#     if ULONG_LONG_MAX == 18446744073709551615ull
#       define BRG_UI64
#       define li_64(h) 0x##h##ull
        typedef unsigned long long uint_64t;
#     endif
#   endif
#endif

#if !defined( BRG_UI64 )
#   if defined( NEED_UINT_64T )
#     error Please define uint_64t as an unsigned 64 bit type in brg_types.h
#   endif
#endif

#ifndef RETURN_VALUES
#   define RETURN_VALUES
#   if defined( DLL_EXPORT )
#     if defined( _MSC_VER ) || defined ( __INTEL_COMPILER )
#       define VOID_RETURN    __declspec( dllexport ) void __stdcall
#       define INT_RETURN     __declspec( dllexport ) int  __stdcall
#     elif defined( __GNUC__ )
#       define VOID_RETURN    __declspec( __dllexport__ ) void
#       define INT_RETURN     __declspec( __dllexport__ ) int
#     else
#       error Use of the DLL is only available on the Microsoft, Intel and GCC compilers
#     endif
#   elif defined( DLL_IMPORT )
#     if defined( _MSC_VER ) || defined ( __INTEL_COMPILER )
#       define VOID_RETURN    __declspec( dllimport ) void __stdcall
#       define INT_RETURN     __declspec( dllimport ) int  __stdcall
#     elif defined( __GNUC__ )
#       define VOID_RETURN    __declspec( __dllimport__ ) void
#       define INT_RETURN     __declspec( __dllimport__ ) int
#     else
#       error Use of the DLL is only available on the Microsoft, Intel and GCC compilers
#     endif
#   elif defined( __WATCOMC__ )
#     define VOID_RETURN  void __cdecl
#     define INT_RETURN   int  __cdecl
#   else
#     define VOID_RETURN  void
#     define INT_RETURN   int
#   endif
#endif

```

```c
/*      These defines are used to detect and set the memory alignment of pointers.
    Note that offsets are in bytes.

    ALIGN_OFFSET(x,n)                   return the positive or zero offset of
                                the memory addressed by the pointer 'x'
                                from an address that is aligned on an
                                'n' byte boundary ('n' is a power of 2)

    ALIGN_FLOOR(x,n)                    return a pointer that points to memory
                                that is aligned on an 'n' byte boundary
                                and is not higher than the memory address
                                pointed to by 'x' ('n' is a power of 2)

    ALIGN_CEIL(x,n)                             return a pointer that points to memory
                                that is aligned on an 'n' byte boundary
                                and is not lower than the memory address
                                pointed to by 'x' ('n' is a power of 2)
*/

#define ALIGN_OFFSET(x,n)       (((ptrint_t)(x)) & ((n) - 1))
#define ALIGN_FLOOR(x,n)        ((uint_8t*)(x) - ( ((ptrint_t)(x)) & ((n) - 1)))
#define ALIGN_CEIL(x,n)         ((uint_8t*)(x) + (-((ptrint_t)(x)) & ((n) - 1)))

/*  These defines are used to declare buffers in a way that allows
    faster operations on longer variables to be used.  In all these
    defines 'size' must be a power of 2 and >= 8. NOTE that the
    buffer size is in bytes but the type length is in bits

    UNIT_TYPEDEF(x,size)        declares a variable 'x' of length
                                'size' bits

    BUFR_TYPEDEF(x,size,bsize)  declares a buffer 'x' of length 'bsize'
                                bytes defined as an array of variables
```

```
                                each of 'size' bits (bsize must be a
                                multiple of size / 8)

    UNIT_CAST(x,size)           casts a variable to a type of
                                length 'size' bits

    UPTR_CAST(x,size)           casts a pointer to a pointer to a
                                varaiable of length 'size' bits
*/

#define UI_TYPE(size)               uint_##size##t
#define UNIT_TYPEDEF(x,size)        typedef UI_TYPE(size) x
#define BUFR_TYPEDEF(x,size,bsize)  typedef UI_TYPE(size) x[bsize / (
↪size >> 3)]
#define UNIT_CAST(x,size)           ((UI_TYPE(size) )(x))
#define UPTR_CAST(x,size)           ((UI_TYPE(size)*)(x))

#if defined(__cplusplus)
}
#endif

#endif
```

src/Obf/cmac.c

```
/*
---------------------------------------------------------------------------
↪
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights
↪reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied
↪warranties
in respect of its operation, including, but not limited to,
↪correctness
and fitness for purpose.
---------------------------------------------------------------------------
↪
Issue Date: 6/10/2008
*/

```

```c
#include "cmac.h"

#define BLK_ADR_MASK    (BLOCK_SIZE - 1)

void cmac_init(const unsigned char key[], cmac_ctx ctx[1])
{
    memset(ctx, 0, sizeof(cmac_ctx));
    aes_encrypt_key128(key, ctx->aes);
}

void cmac_data(const unsigned char buf[], unsigned long len, cmac_ctx ctx[1])
{   uint_32t cnt = 0, b_pos = ctx->txt_cnt & BLK_ADR_MASK;

    if(!len)
        return;

    if(!((buf - (UI8_PTR(ctx->txt_cbc) + b_pos)) & BUF_ADRMASK))
    {
        while(cnt < len && (b_pos & BUF_ADRMASK))
            UI8_PTR(ctx->txt_cbc)[b_pos++] ^= buf[cnt++];

        while(cnt + BLOCK_SIZE <= len)
        {
            while(cnt + BUF_INC <= len && b_pos <= BLOCK_SIZE - BUF_INC)
            {
                *UNIT_PTR(UI8_PTR(ctx->txt_cbc) + b_pos) ^= *UNIT_PTR(buf + cnt);
                cnt += BUF_INC; b_pos += BUF_INC;
            }

            while(cnt + BLOCK_SIZE <= len)
            {
                aes_ecb_encrypt(UI8_PTR(ctx->txt_cbc), UI8_PTR(ctx->txt_cbc), AES_BLOCK_SIZE, ctx->aes);
                xor_block_aligned(ctx->txt_cbc, ctx->txt_cbc, buf + cnt);
                cnt += BLOCK_SIZE;
            }
        }
    }
    else
    {
        while(cnt < len && b_pos < BLOCK_SIZE)
            UI8_PTR(ctx->txt_cbc)[b_pos++] ^= buf[cnt++];

        while(cnt + BLOCK_SIZE <= len)
        {
```

```
                aes_ecb_encrypt(UI8_PTR(ctx->txt_cbc), UI8_PTR(ctx->
↪txt_cbc), AES_BLOCK_SIZE, ctx->aes);
                xor_block(ctx->txt_cbc, ctx->txt_cbc, buf + cnt);
                cnt += BLOCK_SIZE;
            }
        }

        while(cnt < len)
        {
            if(b_pos == BLOCK_SIZE)
            {
                aes_ecb_encrypt(UI8_PTR(ctx->txt_cbc), UI8_PTR(ctx->
↪txt_cbc), AES_BLOCK_SIZE, ctx->aes);
                b_pos = 0;
            }
            UI8_PTR(ctx->txt_cbc)[b_pos++] ^= buf[cnt++];
        }

        ctx->txt_cnt += cnt;
}

static const unsigned char c_xor[4] = { 0x00, 0x87, 0x0e, 0x89 };

static void gf_mulx(uint_8t pad[BLOCK_SIZE])
{   int i, t = pad[0] >> 7;

    for(i = 0; i < BLOCK_SIZE - 1; ++i)
        pad[i] = (pad[i] << 1) | (pad[i + 1] >> 7);
    pad[BLOCK_SIZE - 1] = (pad[BLOCK_SIZE - 1] << 1) ^ c_xor[t];
}

void gf_mulx2(uint_8t pad[BLOCK_SIZE])
{   int i, t = pad[0] >> 6;

    for(i = 0; i < BLOCK_SIZE - 1; ++i)
        pad[i] = (pad[i] << 2) | (pad[i + 1] >> 6);
    pad[BLOCK_SIZE - 2] ^= (t >> 1);
    pad[BLOCK_SIZE - 1] = (pad[BLOCK_SIZE - 1] << 2) ^ c_xor[t];
}

void cmac_end(unsigned char auth_tag[], cmac_ctx ctx[1])
{   buf_type pad;
    int i;

    memset(pad, 0, sizeof(pad));
    aes_ecb_encrypt(UI8_PTR(pad), UI8_PTR(pad), AES_BLOCK_SIZE, ctx->
↪aes);
    i = ctx->txt_cnt & BLK_ADR_MASK;
    if(ctx->txt_cnt == 0 || i)
```

```
    {
        UI8_PTR(ctx->txt_cbc)[i] ^= 0x80;
        gf_mulx2(UI8_PTR(pad));
    }
    else
        gf_mulx(UI8_PTR(pad));

    xor_block_aligned(pad, pad, ctx->txt_cbc);
    aes_ecb_encrypt(UI8_PTR(pad), UI8_PTR(pad), AES_BLOCK_SIZE, ctx->
↪aes);

    for(i = 0; i < BLOCK_SIZE; ++i)
        auth_tag[i] = UI8_PTR(pad)[i];
}
```

src/Obf/cmac.h

```
/*
---------------------------------------------------------------------------
↪
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights
↪reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied
↪warranties
in respect of its operation, including, but not limited to,
↪correctness
and fitness for purpose.
---------------------------------------------------------------------------
↪
Issue Date: 6/10/2008
*/

#ifndef CMAC_AES_H
#define CMAC_AES_H

#if !defined( UNIT_BITS )
#   if 1
#     define UNIT_BITS 64
#   elif 0
#     define UNIT_BITS 32
```

```
#   else
#     define UNIT_BITS  8
#   endif
#endif

#include <string.h>
#include "aes.h"
#include "mode_hdr.h"

UNIT_TYPEDEF(buf_unit, UNIT_BITS);
BUFR_TYPEDEF(buf_type, UNIT_BITS, AES_BLOCK_SIZE);

#if defined(__cplusplus)
extern "C"
{
#endif

#define BLOCK_SIZE  AES_BLOCK_SIZE

typedef struct
{
    buf_type        txt_cbc;
    aes_encrypt_ctx aes[1];                 /* AES encryption context             */
    uint_32t        txt_cnt;
} cmac_ctx;

void cmac_init( const unsigned char key[],  /* the encryption key                 */
                cmac_ctx ctx[1] );          /* the OMAC context                    */

void cmac_data( const unsigned char buf[],        /* the data buffer                */
                unsigned long len,          /* the length of this block (bytes) */
                cmac_ctx ctx[1] );          /* the OMAC context                    */

void cmac_end( unsigned char auth_tag[],    /* the encryption key                 */
               cmac_ctx ctx[1] );           /* the OMAC context                    */

#if defined(__cplusplus)
}
#endif

#endif
```

src/Obf/mode_hdr.h

```
/*
---------------------------------------------------------------------------
Copyright (c) 1998-2010, Brian Gladman, Worcester, UK. All rights reserved.

The redistribution and use of this software (with or without changes)
is allowed without the payment of fees or royalties provided that:

  source code distributions include the above copyright notice, this
  list of conditions and the following disclaimer;

  binary distributions include the above copyright notice, this list
  of conditions and the following disclaimer in their documentation.

This software is provided 'as is' with no explicit or implied warranties
in respect of its operation, including, but not limited to, correctness
and fitness for purpose.
---------------------------------------------------------------------------
Issue Date: 07/10/2010

This header file is an INTERNAL file which supports mode implementation
*/

#ifndef _MODE_HDR_H
#define _MODE_HDR_H

#include <string.h>
#include <limits.h>

#include "brg_endian.h"

/*  This define sets the units in which buffers are processed.  This code
    can provide significant speed gains if buffers can be processed in
    32 or 64 bit chunks rather than in bytes.  This define sets the units
    in which buffers will be accessed if possible
*/
#if !defined( UNIT_BITS )
#  if PLATFORM_BYTE_ORDER == IS_BIG_ENDIAN
#    if 0
#      define UNIT_BITS  32
```

```c
#    elif 1
#      define UNIT_BITS  64
#    endif
#  elif defined( _WIN64 )
#    define UNIT_BITS 64
#  else
#    define UNIT_BITS 32
#  endif
#endif

#if UNIT_BITS == 64 && !defined( NEED_UINT_64T )
#  define NEED_UINT_64T
#endif

#include "brg_types.h"

/*  Use of inlines is preferred but code blocks can also be expanded inline
    using 'defines'.  But the latter approach will typically generate a LOT
    of code and is not recommended.
*/
#if 1 && !defined( USE_INLINING )
#  define USE_INLINING
#endif

#if defined( _MSC_VER )
#  if _MSC_VER >= 1400
#    include <stdlib.h>
#    include <intrin.h>
#    pragma intrinsic(memset)
#    pragma intrinsic(memcpy)
#    define rotl32        _rotl
#    define rotr32        _rotr
#    define rotl64        _rotl64
#    define rotr64        _rotl64
#    define bswap_16(x)   _byteswap_ushort(x)
#    define bswap_32(x)   _byteswap_ulong(x)
#    define bswap_64(x)   _byteswap_uint64(x)
#  else
#    define rotl32 _lrotl
#    define rotr32 _lrotr
#  endif
#endif

#if defined( USE_INLINING )
#  if defined( _MSC_VER )
#    define mh_decl __inline
#  elif defined( __GNUC__ ) || defined( __GNU_LIBRARY__ )
```

```
#    define mh_decl static inline
#  else
#    define mh_decl static
#  endif
#endif

#if defined(__cplusplus)
extern "C" {
#endif

#define  UI8_PTR(x)     UPTR_CAST(x,  8)
#define UI16_PTR(x)     UPTR_CAST(x, 16)
#define UI32_PTR(x)     UPTR_CAST(x, 32)
#define UI64_PTR(x)     UPTR_CAST(x, 64)
#define UNIT_PTR(x)     UPTR_CAST(x, UNIT_BITS)

#define  UI8_VAL(x)     UNIT_CAST(x,  8)
#define UI16_VAL(x)     UNIT_CAST(x, 16)
#define UI32_VAL(x)     UNIT_CAST(x, 32)
#define UI64_VAL(x)     UNIT_CAST(x, 64)
#define UNIT_VAL(x)     UNIT_CAST(x, UNIT_BITS)

#define BUF_INC          (UNIT_BITS >> 3)
#define BUF_ADRMASK     ((UNIT_BITS >> 3) - 1)

#define rep2_u2(f,r,x)    f( 0,r,x); f( 1,r,x)
#define rep2_u4(f,r,x)    f( 0,r,x); f( 1,r,x); f( 2,r,x); f( 3,r,x)
#define rep2_u16(f,r,x)   f( 0,r,x); f( 1,r,x); f( 2,r,x); f( 3,r,x);
↪ \
                          f( 4,r,x); f( 5,r,x); f( 6,r,x); f( 7,r,x);
↪ \
                          f( 8,r,x); f( 9,r,x); f(10,r,x); f(11,r,x);
↪ \
                          f(12,r,x); f(13,r,x); f(14,r,x); f(15,r,x)

#define rep2_d2(f,r,x)    f( 1,r,x); f( 0,r,x)
#define rep2_d4(f,r,x)    f( 3,r,x); f( 2,r,x); f( 1,r,x); f( 0,r,x)
#define rep2_d16(f,r,x)   f(15,r,x); f(14,r,x); f(13,r,x); f(12,r,x);
↪ \
                          f(11,r,x); f(10,r,x); f( 9,r,x); f( 8,r,x);
↪ \
                          f( 7,r,x); f( 6,r,x); f( 5,r,x); f( 4,r,x);
↪ \
                          f( 3,r,x); f( 2,r,x); f( 1,r,x); f( 0,r,x)

#define rep3_u2(f,r,x,y,c)  f( 0,r,x,y,c); f( 1,r,x,y,c)
#define rep3_u4(f,r,x,y,c)  f( 0,r,x,y,c); f( 1,r,x,y,c); f( 2,r,x,y,
↪c); f( 3,r,x,y,c)
#define rep3_u16(f,r,x,y,c) f( 0,r,x,y,c); f( 1,r,x,y,c); f( 2,r,x,y,
```

```
c); f( 3,r,x,y,c); \
                               f( 4,r,x,y,c); f( 5,r,x,y,c); f( 6,r,x,y,c); f( 7,r,x,y,c); \
                               f( 8,r,x,y,c); f( 9,r,x,y,c); f(10,r,x,y,c); f(11,r,x,y,c); \
                               f(12,r,x,y,c); f(13,r,x,y,c); f(14,r,x,y,c); f(15,r,x,y,c)

#define rep3_d2(f,r,x,y,c)  f( 1,r,x,y,c); f( 0,r,x,y,c)
#define rep3_d4(f,r,x,y,c)  f( 3,r,x,y,c); f( 2,r,x,y,c); f( 1,r,x,y,c); f( 0,r,x,y,c)
#define rep3_d16(f,r,x,y,c) f(15,r,x,y,c); f(14,r,x,y,c); f(13,r,x,y,c); f(12,r,x,y,c); \
                               f(11,r,x,y,c); f(10,r,x,y,c); f( 9,r,x,y,c); f( 8,r,x,y,c); \
                               f( 7,r,x,y,c); f( 6,r,x,y,c); f( 5,r,x,y,c); f( 4,r,x,y,c); \
                               f( 3,r,x,y,c); f( 2,r,x,y,c); f( 1,r,x,y,c); f( 0,r,x,y,c)

/* function pointers might be used for fast XOR operations */

typedef void (*xor_function)(void* r, const void* p, const void* q);

/* left and right rotates on 32 and 64 bit variables */

#if !defined( rotl32 )  /* NOTE: 0 <= n <= 32 ASSUMED */
mh_decl uint_32t rotl32(uint_32t x, int n)
{
    return (((x) << n) | ((x) >> (32 - n)));
}
#endif

#if !defined( rotr32 )  /* NOTE: 0 <= n <= 32 ASSUMED */
mh_decl uint_32t rotr32(uint_32t x, int n)
{
    return (((x) >> n) | ((x) << (32 - n)));
}
#endif

#if UNIT_BITS == 64 && !defined( rotl64 )  /* NOTE: 0 <= n <= 64 ASSUMED */
mh_decl uint_64t rotl64(uint_64t x, int n)
{
    return (((x) << n) | ((x) >> (64 - n)));
}
#endif

#if UNIT_BITS == 64 && !defined( rotr64 )  /* NOTE: 0 <= n <= 64
```

```
↪ASSUMED */
mh_decl uint_64t rotr64(uint_64t x, int n)
{
    return (((x) >> n) | ((x) << (64 - n)));
}
#endif

/* byte order inversions for 16, 32 and 64 bit variables */

#if !defined(bswap_16)
mh_decl uint_16t bswap_16(uint_16t x)
{
    return (uint_16t)((x >> 8) | (x << 8));
}
#endif

#if !defined(bswap_32)
mh_decl uint_32t bswap_32(uint_32t x)
{
    return ((rotr32((x), 24) & 0x00ff00ff) | (rotr32((x), 8) & 0
↪xff00ff00));
}
#endif

#if UNIT_BITS == 64 && !defined(bswap_64)
mh_decl uint_64t bswap_64(uint_64t x)
{
    return bswap_32((uint_32t)(x >> 32)) | ((uint_64t)bswap_32((
↪uint_32t)x) << 32);
}
#endif

/* support for fast aligned buffer move, xor and byte swap operations
↪ -
   source and destination buffers for move and xor operations must
↪not
   overlap, those for byte order revesal must either not overlap or
   must be identical
*/
#define f_copy(n,p,q)     p[n] = q[n]
#define f_xor(n,r,p,q,c)  r[n] = c(p[n] ^ q[n])

mh_decl void copy_block(void* p, const void* q)
{
    memcpy(p, q, 16);
}

mh_decl void copy_block_aligned(void *p, const void *q)
{
```

```c
#if UNIT_BITS == 8
    memcpy(p, q, 16);
#elif UNIT_BITS == 32
    rep2_u4(f_copy,UNIT_PTR(p),UNIT_PTR(q));
#else
    rep2_u2(f_copy,UNIT_PTR(p),UNIT_PTR(q));
#endif
}

mh_decl void xor_block(void *r, const void* p, const void* q)
{
    rep3_u16(f_xor, UI8_PTR(r), UI8_PTR(p), UI8_PTR(q), UI8_VAL);
}

mh_decl void xor_block_aligned(void *r, const void *p, const void *q)
{
#if UNIT_BITS == 8
    rep3_u16(f_xor, UNIT_PTR(r), UNIT_PTR(p), UNIT_PTR(q), UNIT_VAL);
#elif UNIT_BITS == 32
    rep3_u4(f_xor, UNIT_PTR(r), UNIT_PTR(p), UNIT_PTR(q), UNIT_VAL);
#else
    rep3_u2(f_xor, UNIT_PTR(r), UNIT_PTR(p), UNIT_PTR(q), UNIT_VAL);
#endif
}

/* byte swap within 32-bit words in a 16 byte block; don't move 32-bit words */
mh_decl void bswap32_block(void *d, const void* s)
{
#if UNIT_BITS == 8
    uint_8t t;
    t = UNIT_PTR(s)[ 0]; UNIT_PTR(d)[ 0] = UNIT_PTR(s)[ 3]; UNIT_PTR(d)[ 3] = t;
    t = UNIT_PTR(s)[ 1]; UNIT_PTR(d)[ 1] = UNIT_PTR(s)[ 2]; UNIT_PTR(d)[ 2] = t;
    t = UNIT_PTR(s)[ 4]; UNIT_PTR(d)[ 4] = UNIT_PTR(s)[ 7]; UNIT_PTR(d)[ 7] = t;
    t = UNIT_PTR(s)[ 5]; UNIT_PTR(d)[ 5] = UNIT_PTR(s)[ 6]; UNIT_PTR(d) [6] = t;
    t = UNIT_PTR(s)[ 8]; UNIT_PTR(d)[ 8] = UNIT_PTR(s)[11]; UNIT_PTR(d)[12] = t;
    t = UNIT_PTR(s)[ 9]; UNIT_PTR(d)[ 9] = UNIT_PTR(s)[10]; UNIT_PTR(d)[10] = t;
    t = UNIT_PTR(s)[12]; UNIT_PTR(d)[12] = UNIT_PTR(s)[15]; UNIT_PTR(d)[15] = t;
    t = UNIT_PTR(s)[13]; UNIT_PTR(d)[ 3] = UNIT_PTR(s)[14]; UNIT_PTR(d)[14] = t;
#elif UNIT_BITS == 32
    UNIT_PTR(d)[0] = bswap_32(UNIT_PTR(s)[0]); UNIT_PTR(d)[1] =
```

```
bswap_32(UNIT_PTR(s)[1]);
    UNIT_PTR(d)[2] = bswap_32(UNIT_PTR(s)[2]); UNIT_PTR(d)[3] = bswap_32(UNIT_PTR(s)[3]);
#else
    UI32_PTR(d)[0] = bswap_32(UI32_PTR(s)[0]); UI32_PTR(d)[1] = bswap_32(UI32_PTR(s)[1]);
    UI32_PTR(d)[2] = bswap_32(UI32_PTR(s)[2]); UI32_PTR(d)[3] = bswap_32(UI32_PTR(s)[3]);
#endif
}

/* byte swap within 64-bit words in a 16 byte block; don't move 64-bit words */
mh_decl void bswap64_block(void *d, const void* s)
{
#if UNIT_BITS == 8
    uint_8t t;
    t = UNIT_PTR(s)[ 0]; UNIT_PTR(d)[ 0] = UNIT_PTR(s)[ 7]; UNIT_PTR(d)[ 7] = t;
    t = UNIT_PTR(s)[ 1]; UNIT_PTR(d)[ 1] = UNIT_PTR(s)[ 6]; UNIT_PTR(d)[ 6] = t;
    t = UNIT_PTR(s)[ 2]; UNIT_PTR(d)[ 2] = UNIT_PTR(s)[ 5]; UNIT_PTR(d)[ 5] = t;
    t = UNIT_PTR(s)[ 3]; UNIT_PTR(d)[ 3] = UNIT_PTR(s)[ 3]; UNIT_PTR(d) [3] = t;
    t = UNIT_PTR(s)[ 8]; UNIT_PTR(d)[ 8] = UNIT_PTR(s)[15]; UNIT_PTR(d)[15] = t;
    t = UNIT_PTR(s)[ 9]; UNIT_PTR(d)[ 9] = UNIT_PTR(s)[14]; UNIT_PTR(d)[14] = t;
    t = UNIT_PTR(s)[10]; UNIT_PTR(d)[10] = UNIT_PTR(s)[13]; UNIT_PTR(d)[13] = t;
    t = UNIT_PTR(s)[11]; UNIT_PTR(d)[11] = UNIT_PTR(s)[12]; UNIT_PTR(d)[12] = t;
#elif UNIT_BITS == 32
    uint_32t t;
    t = bswap_32(UNIT_PTR(s)[0]); UNIT_PTR(d)[0] = bswap_32(UNIT_PTR(s)[1]); UNIT_PTR(d)[1] = t;
    t = bswap_32(UNIT_PTR(s)[2]); UNIT_PTR(d)[2] = bswap_32(UNIT_PTR(s)[2]); UNIT_PTR(d)[3] = t;
#else
    UNIT_PTR(d)[0] = bswap_64(UNIT_PTR(s)[0]);  UNIT_PTR(d)[1] = bswap_64(UNIT_PTR(s)[1]);
#endif
}

mh_decl void bswap128_block(void *d, const void* s)
{
#if UNIT_BITS == 8
    uint_8t t;
```

```
    t = UNIT_PTR(s)[0]; UNIT_PTR(d)[0] = UNIT_PTR(s)[15]; UNIT_PTR(d)[15] = t;
    t = UNIT_PTR(s)[1]; UNIT_PTR(d)[1] = UNIT_PTR(s)[14]; UNIT_PTR(d)[14] = t;
    t = UNIT_PTR(s)[2]; UNIT_PTR(d)[2] = UNIT_PTR(s)[13]; UNIT_PTR(d)[13] = t;
    t = UNIT_PTR(s)[3]; UNIT_PTR(d)[3] = UNIT_PTR(s)[12]; UNIT_PTR(d)[12] = t;
    t = UNIT_PTR(s)[4]; UNIT_PTR(d)[4] = UNIT_PTR(s)[11]; UNIT_PTR(d)[11] = t;
    t = UNIT_PTR(s)[5]; UNIT_PTR(d)[5] = UNIT_PTR(s)[10]; UNIT_PTR(d)[10] = t;
    t = UNIT_PTR(s)[6]; UNIT_PTR(d)[6] = UNIT_PTR(s)[ 9]; UNIT_PTR(d)[ 9] = t;
    t = UNIT_PTR(s)[7]; UNIT_PTR(d)[7] = UNIT_PTR(s)[ 8]; UNIT_PTR(d)[ 8] = t;
#elif UNIT_BITS == 32
    uint_32t t;
    t = bswap_32(UNIT_PTR(s)[0]); UNIT_PTR(d)[0] = bswap_32(UNIT_PTR(s)[3]); UNIT_PTR(d)[3] = t;
    t = bswap_32(UNIT_PTR(s)[1]); UNIT_PTR(d)[1] = bswap_32(UNIT_PTR(s)[2]); UNIT_PTR(d)[2] = t;
#else
    uint_64t t;
    t = bswap_64(UNIT_PTR(s)[0]); UNIT_PTR(d)[0] = bswap_64(UNIT_PTR(s)[1]); UNIT_PTR(d)[1] = t;
#endif
}

/* platform byte order to big or little endian order for 16, 32 and 64 bit variables */

#if PLATFORM_BYTE_ORDER == IS_BIG_ENDIAN

#  define uint_16t_to_le(x) (x) = bswap_16((x))
#  define uint_32t_to_le(x) (x) = bswap_32((x))
#  define uint_64t_to_le(x) (x) = bswap_64((x))
#  define uint_16t_to_be(x)
#  define uint_32t_to_be(x)
#  define uint_64t_to_be(x)

#else

#  define uint_16t_to_le(x)
#  define uint_32t_to_le(x)
#  define uint_64t_to_le(x)
#  define uint_16t_to_be(x) (x) = bswap_16((x))
#  define uint_32t_to_be(x) (x) = bswap_32((x))
#  define uint_64t_to_be(x) (x) = bswap_64((x))
```

```

#endif

#if defined(__cplusplus)
}
#endif

#endif
```