Viper — Final Report

COSC 4853 Network Security Project

Nicholas Capo - Nicholas.Capo@Gmail.com

April 21, 2012

1 Draft Literature Review

1.1 Overview

"Serpent is a symmetric key block cipher which was a finalist in the Advanced Encryption Standard (AES) contest, where it came second to Rijndael. Serpent was designed by Ross Anderson, Eli Biham, and Lars Knudsen."[11]

1.2 Design Methodology

A significant portion of Serpent was based on the extensive cryptanalysis completed on the Data Encryption Standard (DES). [6] This allows the authors to design a a stronger cipher while still maintaining the known advantages of DES. The original submission to AES actually used the same S-Boxes as DES[6, p. 5], but later it was determined that suitably complex S-Boxes could be created using a deterministic method (thereby avoiding the potential for built-in trapdoors). [6, pp. 7, 15]

1.3 Parallelism

By using a bit-slice implementation, Serpent can be efficiently implemented on a "processor with two 32-bit integer ALUs (such as the popular Intel MMX series)".[6, p. 2] From Wikipedia: "Serpent was designed so that all operations can be executed in parallel, using 32 1-bit slices. This maximizes [the] parallelism [of the algorithm]"[11]

1.4 Conservative Approach

Using the cryptanalysis of DES as a basis the authors of Serpent took a conservative approach to the design of the algorithm.[11] Since it was expected (and we now know) that AES would be employed for many years after the competition, Serpent has been designed with an eye to the future. The authors believe that as few as 16 rounds of permutation-substitution would be secure for many years, however the design proposes 32 rounds as a precaution against feature advances in cryptanalysis. [6, pp. 4, 8]

1.5 Reference Implementations

During the AES selection process, several references implementations were developed in several languages: Specifically, Ada, Assembler, C, Java, and Python.[6, p. 16] Several of these implementations are available for download from the Serpent Homepage [1].

1.6 Patents and Licensing

The authors of Serpent have applied for a U.K. Patent (Application 9722798.9. Filed October 30, 1997) [3] but to quote the Serpent Homepage:

Serpent is now completely in the public domain, and we impose no restrictions on its use. This was announced on the 21st August at the First AES Candidate Conference. The optimised implementations in the submission package are now under the General Public License (GPL), although some comments in the code still say otherwise. ([1])

2 Problem and Proposed Solution

2.1 Problem Statement

According to [11]: "Serpent was designed so that all operations can be executed in parallel, using 32 1-bit slices. This maximizes [the] parallelism [of the algorithm]". However, the context of these (and other) statements seem to imply that it would only be efficient to parallelize Serpent in hardware (or very close to hardware, e.g. Assembly). But the efficiency gains of a parallelized implementation in software are not addressed.

2.2 Proposed Solution

Construct a cipher-text compatible implementation of the Serpent Algorithm in both C and Python. Each implementation shall be capable of encryption and decryption using a single thread[1] as well as 32 parallel threads as described in [6]. These implementations can then be compared for speed and efficiency, in threaded and non-threaded modes, and the results analyzed to determine if there is any advantage to implementing software parallelism in Serpent.

[1] `pThread` in the case of C, and the `multiprocessing` module in the case of Python (see the note at http://docs.python.org/library/threading.html for why `multiprocessing` was chosen over `threading`)

3 Implementation

3.1 Requirements Analysis

3.1.1 Language

1. The program shall be referred to herein as *Viper*
2. The overall design shall follow the description in [6]
3. One version of the program shall be produced using the C language
4. One version of the program shall be produced using the Python language

3.1.2 Binaries

1. Each version shall be compiled into two binaries (`viper` and `viperBlockTest`) with the following usage:
   (a) `viper [ -h | --help ] [ -e | -d | --encrypt | --decrypt ] [ -t | --threads NUM ] [ -k | --key KEY ]`
   (b) `viperBlockTest [ -h | --help ] [ -e | -d | --encrypt | --decrypt ] [ -k | --key KEY ] input_block`

3.1.3 Modules

1. Each implementation shall be broken into at least three modules: (See 3.1.6 for details of the threading requirements)
   (a) a single–threaded `main()`
   (b) a multi–threaded `main()`
   (c) a `viperCrypt` module, containing the implementation of the cipher specification itself.

3.1.4 Input/Output

1. `viper` shall expect input on `stdin`, and generate output on `stdout`
2. `viperBlockTest` shall expect a single block of 32 hexadecimal values as the last argument on the command line
3. `viper` shall be the general case of `viperBlockTest` and shall encrypt or decrypt until reaching end–of–input
4. All errors and help texts shall be written to `stderr`

3.1.5 Compatibility

1. Each version of `viper` shall be ciphertext compatible with the reference implementation of `Serpent` [5, 10]

3.1.6 Threading

1. Each version of `viper` shall implement a single–threaded mode
2. Each version of `viper` shall implement a multi–threaded mode, using 32 threads

3.2 Design

3.2.1 Overview

The software shall be designed using a primarily functional approach. The core of the algorithm shall be created in a module named `viperCrypt`. Normal interaction with the module shall occur by calling the `crypt()` function (See Dataflow Diagram) and passing the user key, the plain– or cipher–text, and a flag, which indicates whether to encrypt or decrypt. The `crypt()` function shall return the result of the encryption or decryption.

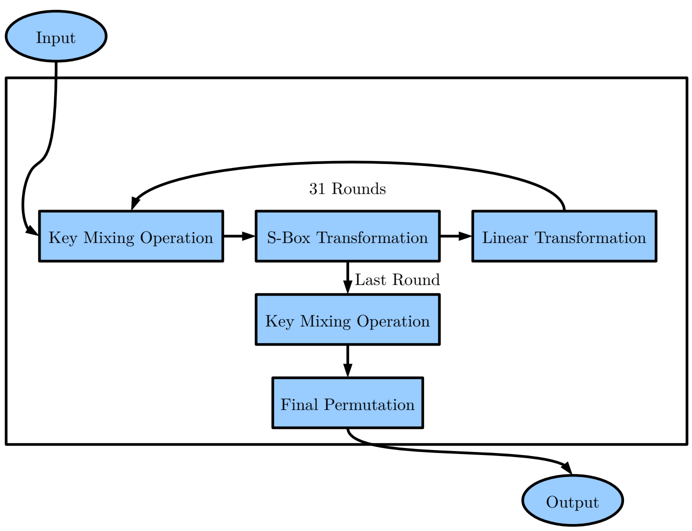

Figure 1: Dataflow Diagram

3.2.2 Multi–Threading

The multi–threaded version of `Viper` shall be implemented as 32 threads (See Threaded Dataflow Diagram), where each thread consists of `viperCrypt.crypt()` operating on a separate block of input data.

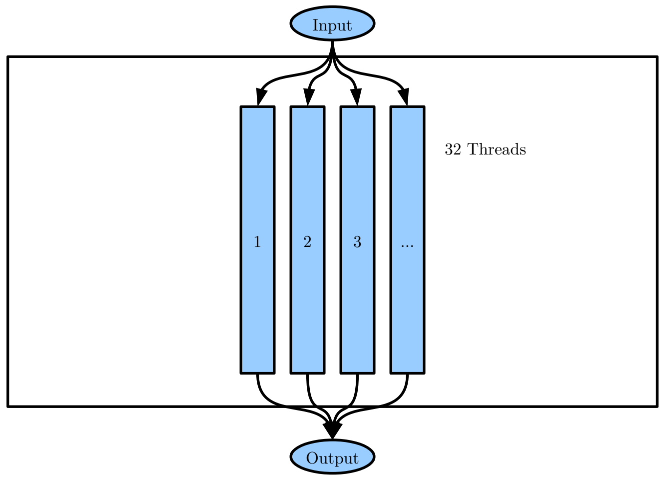

Figure 2: Threaded Dataflow Diagram

3.3 Implementation Results

3.3.1 Environment

The implementation was constructed and tested using the following environment on an x86 architecture:

- Ubuntu Linux (version 10.04.4 LTS)
- Debian Linux (version Testing/“Weezy”)
- Python (version 2.6.5)
- GCC (version 4.4.3)
- GNU Make (version 3.81)

It is expected that the implementation will be compatible with any platform that runs Python and/or C.

3.3.2 Source Files

- C
  - `sbox.h`
  - `viperBlockTest.c`
  - `viperCrypt.c`

- viper.c

- Python
  - sbox.py
  - viperBlockTest.py
  - viperCrypt.py
  - viper.py

3.3.3 Internal Dependencies

Each version of `viper` depends on the `viperCrypt` module with in turn depends on the `sbox` module.

3.3.4 External Dependencies

C Only the standard C libraries were used.

Python Each of the following Python modules were imported into one or more source files:

- `argparse`
- `sys`
- `print_function` [2]

3.3.5 Build Instructions

C Using the provided `Makefile` should be sufficient. However the following commands may be used as well:

- `gcc -Wall viper.c -o viper.exe`
- `gcc -Wall viperBlockTest.c -o viperBlockTest.exe`

Python No building is necessary, all required compilation will occur as a result of running `python viper.py`.

3.4 Test Methodology

3.4.1 Unit Tests

1. Unit tests, ad-hoc tests, and other small tests shall be used to confirm the basic operation of functions etc.

[2] This function was imported from the `future` module to provide Python 3.x printing features

3.4.2 Single Block Acceptance Tests

1. `viperBlockTest` shall be used in conjunction with the *Known Answer Test, and Monte Carlo Test* in [5] to confirm the correctness of the simple cipher implementation.

3.4.3 Multi–Block Acceptance Tests

1. The single– and multi–threaded versions of `viper` shall be used in conjunction with the reference Implementations in [5, 10] to confirm the correctness of the complete cipher implementation, and that no errors have been introduced in the multi–threaded implementation.

3.4.4 Speed Tests

1. The single– and multi–threaded versions of `viper` shall be used to encrypt and decrypt files of various sizes and the encryption and decryption times recorded for comparison.

2. The following Speed Tests shall be used:

   (a) A zero–filled file in the following sizes

      i. 1B
      ii. 32B
      iii. 100B
      iv. 500B
      v. 1KB
      vi. 32KB
      vii. 100KB
      viii. 500KB
      ix. 1MB
      x. 32MB
      xi. 100MB
      xii. 500MB
      xiii. 1GB

   (b) Randomly generated files in the following sizes

      i. 1B
      ii. 32B
      iii. 100B
      iv. 500B
      v. 1KB
      vi. 32KB
      vii. 100KB
      viii. 500KB

ix. 1MB
x. 32MB

3. Each test shall be run no less than three times and the results averaged.

4 Results

[Note: At the current time we have not completed the implementation of the cipher. Any results which would have appeared here must necessarily be delayed until the completion of the implementation.]

5 Conclusion

> There is an enormous difference between a mathematical algorithm and its concrete implementation in hardware or software. Cryptographic system designs are fragile. Just because a protocol is logically secure doesn't mean it will stay secure when a designer starts defining message structures and passing bits around. Close isn't close enough; these systems must be implemented exactly, perfectly, or they will fail. —Bruce Schneier ([9])

Implementing an encryption algorithm is hard to do correctly. No we take it back. It's just plain hard. Even more so for a "normal" developer[3]. We think there are two main reasons why implementing an encryption algorithm is difficult for an average developer:

1. Cryptographers talk in math, and
2. Pseudo code isn't an implementation

5.1 Cryptographers talk in math

Most cryptographers explain encryption in terms of math. This is understandable, first because math is the domain of provability, and second because, at its heart, Computer Science *is* Math. But most developers[4] understand programming better than they understand math. This leaves a large gulf for miscommunication.

5.1.1 Math doesn't have types

Is this a correct quotation?

> Computer Science is Math with Types
> —Dr. Jay-Evans Tevis, Ph.D., LeTourneau University

When the author of an encryption algorithm describes its operation, they frequently neglect to describe the data types of the objects they are using, leading to fundamental questions being raised by the developer: "Is that an array of bytes, or a string?" "Is this an integer or a character?" "How do I perform modulo on an array of strings?"

5.2 Pseudo-code isn't an implementation

The second major difficulty in implementing an encryption algorithm is that the authors provide pseudo-code. The problem here is that the pseudo-code is typically incomplete or fragmented, provides no type information to the developer, and frequently contains errors or inconsistencies. In these cases the pseudo-code is actually a hindrance rather than a help to the implementation effort.

[3] i.e. not a cryptographer.

[4] Specifically the author of this paper.

The solution to this is for the cryptographer to work closely with a developer to create a reference implementation. In this way the intent of the cryptographer can be better understood.

5.3 Lessons Learned

With very few exceptions, and unless you are a well trained cryptographer, you should abide by the following three rules:

Don't write your own encryption algorithm: it will be insecure

Don't re-implement a known algorithm: your implementation will be insecure

Do use well-tested implementations of well-tested algorithms

> Building a secure cryptographic system is easy to do badly, and very difficult to do well. Unfortunately, most people can't tell the difference. In other areas of computer science, functionality serves to differentiate the good from the bad: a good compression algorithm will work better than a bad one; a bad compression program will look worse in feature-comparison charts. Cryptography is different. Just because an encryption program works doesn't mean it is secure. What happens with most products is that someone reads Applied Cryptography, chooses an algorithm and protocol, tests it to make sure it works, and thinks he's done. He's not. Functionality does not equal quality, and no amount of beta testing will ever reveal a security flaw. Too many products are merely "buzzword compliant"; they use secure cryptography, but they are not secure. —Bruce Schneier ([8])

5.4 Future Work

- Complete implementation of Serpent as specified above
- Analyze runtime performance of Clyther Implementation (Python OpenCL bindings)
- Research method to allow execution and testing of pseudo-code for use by textbook authors

References

[1] Ross J. Anderson. *Serpent Home Page*. [Online; accessed 26-January-2012]. URL: https://www.cl.cam.ac.uk/~rja14/serpent.html (cit. on pp. 2, 3).

[2] Eli Biham. *Serpent - A New Block Cipher Proposal for AES*. [Online; accessed 18-February-2012]. URL: http://www.cs.technion.ac.il/~biham/Reports/Serpent/.

[3] David Hopwood david.hopwood@zetnet.co.uk. *SCAN – Standard Cryptographic Algorithm Naming*. [Online; accessed 26-January-2012]. 2002. URL: http://www.users.zetnet.co.uk/hopwood/crypto/scan/cs.html#Serpent (cit. on p. 2).

[4] Michael Howard and Keith Brown. *Defend Your Code with Top Ten Security Tips Every Developer Must Know*. [Online; accessed 18-Apr-2012]. 2002. URL: http://msdn.microsoft.com/en-us/magazine/cc188938.aspx#S5.

[5] Ross Anderson, Eli Biham, and Lars Knudsen. *Full submission package, which contains the algorithm specification, a reference implementation in C, an optimised implementation in C and an optimised implementation in Java*. [Online; accessed 18-February-2012]. URL: http://www.cl.cam.ac.uk/~rja14/Papers/serpent.tar.gz (cit. on pp. 5, 9).

[6] Ross Anderson, Eli Biham, and Lars Knudsen. *Serpent: A proposal for the Advanced Encryption Standard*. [Online; accessed 18-February-2012]. URL: http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf (cit. on pp. 2, 4, 5).

[7] Ross Anderson, Eli Biham, and Lars Knudsen. *The Case for Serpent*. [Online; accessed 18-February-2012]. 2000. URL: http://www.cl.cam.ac.uk/~rja14/Papers/serpentcase.pdf.

[8] Bruce Schneier. *Security Pitfalls in Cryptography*. [Online; accessed 18-Apr-2012]. 1998. URL: https://www.schneier.com/essay-028.html (cit. on p. 13).

[9] Bruce Schneier. *Why Cryptography Is Harder Than It Looks*. [Online; accessed 18-Apr-2012]. 1997. URL: https://www.schneier.com/essay-037.html (cit. on p. 12).

[10] Frank Stajano. *Serpent reference implementation*. [Online; accessed 26-January-2012]. URL: https://www.cl.cam.ac.uk/~fms27/serpent/ (cit. on pp. 5, 9).

[11] Wikipedia. *Serpent (cipher) — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-February-2012]. 2012. URL: http://en.wikipedia.org/w/index.php?title=Serpent_(cipher)&oldid=469573199 (cit. on pp. 2, 4).