Cracking DES
Secrets of
How federal
Encryption Research,
agencies
Wiretap Politics
subvert
& Chip Design
privacy

ELECTRONIC FRONTIER FOUNDATION
Cracking DES
Cracking DES
Secrets of Encryption Research, Wiretap Politics & Chip Design

Electronic Frontier Foundation
Cracking DES: Secrets of Encryption Research, Wiretap Politics, and Chip Design
by the Electronic Frontier Foundation

With the exceptions noted, this book and all of its contents are in the public domain. Published in 1998 by the Electronic Frontier Foundation. Printed in the United States of America. No rights reserved. Every part of this book, except as noted below, may be reproduced, in any form or by any means, without permission in writing from the publisher. Because this material is in the public domain, permission to reproduce, use, copy, modify, and distribute this material for any purpose and without fee is hereby granted.

The test-file, bootstrap, and bootstrap2 listings in Chapter 4 are Copyright © 1997 by Network Associates, Inc. These listings may be reproduced in whole or in part without payment of royalties. Chapter 10, Architectural Considerations for Cryptanalytic Hardware, is Copyright ©1996 by the authors, Ian Goldberg and David Wagner. It may not be reproduced without the permission of the authors, who can be reached at iang@cs.berkeley.edu and daw@cs.berkeley.edu. Chapter 11, Efficient DES Key Search: An Update, is Copyright © 1997 by Entrust Technologies. It may be reproduced in whole or in part without payment of royalties. Chapter 9, Breaking One Million DES Keys, is Copyright © 1986. Work done at the University of Leuven, Belgium, and supported by the NFWO, Belgium. It may not be reproduced without the permission of the author, who can be reached at desmedt@cs.uwm.edu.

Distributed by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Printing History:


Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in caps or initial caps.

While many precautions have been taken in the preparation of this book, the publisher and distributor assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

# Table of Contents

**Foreword** ................................................................. ix
**Preface** ........................................................................ xiii

### 1: Overview .................................................................. 1-1

- Politics of Decryption .................................................. 1-1
- Goals ............................................................................ 1-7
- History of DES Cracking ............................................. 1-8
- EFF’s DES Cracker Project ........................................... 1-8
- Architecture ................................................................. 1-9
- Who Else Is Cracking DES? ......................................... 1-16
- What To Do If You Depend On DES ............................ 1-17
- Conclusion ..................................................................... 1-18

### 2: Design for DES Key Search Array ............................ 2-1

- On-Chip Registers ....................................................... 2-1
- Commands ..................................................................... 2-4
- Search Unit Operation .................................................. 2-4
- Sample Programming Descriptions .............................. 2-5
- Scalability and Performance ....................................... 2-9
- Host Computer Software ............................................ 2-9
- Glossary ........................................................................ 2-10
## Table of Contents

### 3: *Design for DES Key Search Array Chip-Level Specification* .......... 3-1
- ASIC Description ........................................................................................................ 3-1
- Board description ........................................................................................................ 3-3
- Read and Write Timing ............................................................................................. 3-5
- Addressing Registers ................................................................................................ 3-7
- All-active Signal ........................................................................................................ 3-7
- ASIC Register Allocation ......................................................................................... 3-8

### 4: *Scanning the Source Code* .............................................................................. 4-1
- The Politics of Cryptographic Source Code .............................................................. 4-1
- The Paper Publishing Exception .............................................................................. 4-2
- Scanning .................................................................................................................... 4-4
- Bootstrapping ............................................................................................................ 4-5

### 5: *Software Source Code* .................................................................................. 5-1

### 6: *Chip Source Code* ......................................................................................... 6-1

### 7: *Chip Simulator Source Code* .......................................................................... 7-1

### 8: *Hardware Board Schematics* .......................................................................... 8-1
- Board Schematics ........................................................................................................ 8-1
- Sun-4/470 backplane modifications .......................................................................... 8-10
- PC Interfaces ............................................................................................................... 8-12
- Errata ......................................................................................................................... 8-13

### 9: *Breaking One Million DES Keys by Yvo Desmedt* .................................... 9-1
- Abstract ....................................................................................................................... 9-1
- Introduction ................................................................................................................ 9-1
- The basic idea ............................................................................................................. 9-2
- Details of such a machine ......................................................................................... 9-2
- Obtained results and remarks .................................................................................. 9-4
- Conclusion .................................................................................................................. 9-4
- Acknowledgement .................................................................................................... 9-5
## Table of Contents

### 10: Architectural Considerations for Cryptanalytic Hardware .. 10-1

- Abstract .................................................................................................................. 10-1
- Introduction .............................................................................................................. 10-1
- Motivation ................................................................................................................ 10-2
- Related work .......................................................................................................... 10-4
- Technical Approach .............................................................................................. 10-6
- Design and Analysis .............................................................................................. 10-8
- Future work ............................................................................................................. 10-23
- Conclusions ............................................................................................................ 10-23
- Acknowledgements ............................................................................................... 10-24
- Availability ............................................................................................................. 10-24
- References ............................................................................................................. 10-24

### 11: Efficient DES Key Search—An Update by Michael J. Wiener 11-1

- Advancing Technology .......................................................................................... 11-2
- Programmable Hardware ....................................................................................... 11-3
- Conclusion ............................................................................................................. 11-4

### 12: Authors ........................................................................................................ 12-1

- The Electronic Frontier Foundation ........................................................................ 12-1
- John Gilmore .......................................................................................................... 12-2
- Cryptography Research ........................................................................................ 12-2
- Paul Kocher ............................................................................................................. 12-3
- Advanced Wireless Technologies ........................................................................... 12-3
Foreword
by Whitfield Diffie

In 1974 the Stanford computer science community ate at Loui's.* As I sat eating one evening in the fall, Butler Lampson approached me, and in the course of inquiring what I was doing, remarked that the IBM Lucifer system was about to be made a national standard. I hadn't known it, and it set me thinking.

My thoughts went as follows:

NSA doesn't want a strong cryptosystem as a national standard, because it is afraid of not being able to read the messages.

On the other hand, if NSA endorses a weak cryptographic system and is discovered, it will get a terrible black eye.

Hints that Butler was correct began to appear and I spent quite a lot of time thinking about this problem over the next few months. It led me to think about trap-door cryptosystems and perhaps ultimately public-key cryptography.

When the Proposed Data Encryption Standard was released on the 17th of March 1975,† I thought I saw what they had done. The basic system might be ok, but the keyspace was on the small side. It would be hard to search, but not impossible. My first estimate was that a machine could be built for $650M that would break DES in a week. I discussed the idea with Marty Hellman and he took it on with a vengeance. Before we were through, the estimated cost had fallen to $20M and the time had declined to a day.‡

* Louis Kao's Hsi-Nan restaurant in Town and Country Village, Palo Alto
† 40 Federal Register 12067
Our paper started a game in the cryptographic community and many papers on searching through DES keys have since been written. About three years after the publication of our paper, Robert Jueneman—then at Satellite Business Systems in McLean, Virginia—wrote "The Data Encryption Standard vs. Exhaustive Search." This opus was substantially more optimistic about the chances for DES breaking. It predicted that by 1985 a half-million dollar investment would get you a DES key every hour and that by 1995, $10 million similarly spent would reduce that time to two seconds, an estimate remarkably close to one made fifteen years later.

A decade later, Yvo Desmedt and Jean-Jacques Quisquater made two contributions, one whimsical, one serious. Using a related "birthday problem" sort of approach, they proposed a machine for attacking many cryptographic problems at a time. Their whimsical suggestion took advantage of the fact that the population of China was about the square root of the size of the DES key space.

The year 1993 brought a watershed. Michael Wiener of Bell-Northern Research (BNR) designed the most solid paper machine yet. It would not be too far off to describe it as a Northern Telecom DMS100 telephone switch, specialized to attacking DES. What made the paper noteworthy was that it used standard Northern Telecom design techniques from the chips to the boards to the cabinets. It anticipated an investment of under a million dollars for a machine that would recover a key every three hours. A provocative aside was the observation that the required budget could be hidden in a director's budget at BNR.

Finally, in 1996, an estimate was prepared by not one or two cryptographers but by a group later, and not entirely sympathetically, called the magnificent seven. This estimate outlined three basic approaches loosely correlated with three levels of resources. At the cheap end was scrounging up time on computers you didn't need to own. In the middle was using programmable logic arrays, possibly PLA machines built for some other purpose such as chip simulation. The high end was the latest refinement of the custom chip approach.

Exhaustive key search is a surprising problem to have enjoyed such popularity. To most people who have considered the problem, it is obvious that a search through $2^{56}$ possibilities is doable if somewhat tedious. If it is a mystery why so many of them, myself included, have worked to refine and solidify their estimates, it is an even greater mystery that in the late 1990s, some people have actually begun to carry out key searches.

At the 1997 annual RSA cryptographic trade show in San Francisco, a prize was announced for cracking a DES cryptogram*. The prize was claimed in five months by a loose consortium using computers scattered around the Internet. It was the most dramatic success so far for an approach earlier applied to factoring and to breaking cryptograms in systems with 40-bit keys.

At the 1998 RSA show, the prize was offered again. This time the prize was claimed in 39 days† a result that actually represents a greater improvement than it appears to. The first key was found after a search of only 25% of the key space; the second was not recovered until the 85% mark. Had the second team been looking for the first key, they would have found it in a month.

These efforts used the magnificent seven’s first approach. No application of the second has yet come to light. This book skips directly to the third. It describes a computer built out of custom chips. A machine that ‘anyone’ can build, from the plans it presents—a machine that can extract DES keys in days at reasonable prices, or hours at high prices. With the appearance of this book and the machine it represents, the game changes forever. It is not a question of whether DES keys can be extracted by exhaustive search; it is a question of how cheaply they can be extracted and for what purposes.

Using a network of general purpose machines that you do not own or control is a perfectly fine way of winning cryptanalytic contests, but it is not a viable way of doing production cryptanalysis. For that, you have to be able to keep your activities to yourself. You need to be able to run on a piece of hardware that you can protect from unwanted scrutiny. This is such a machine. It is difficult to know how many messages have been encrypted with DES in the more than two decades that it has been a standard. Even more difficult is knowing how many of those messages are of enduring interest and how many have already been captured or remain potentially accessible on disks or tapes, but the number, no matter precisely how the question is framed must be large. All of these messages must now be considered to be vulnerable.

* http://www.rsa.com/rsalabs/97challenge/
The vulnerability does not end there, however, for cryptosystems have nine lives. The most convincing argument that DES is insecure would not outweigh the vast investment in DES equipment that has accumulated throughout the world. People will continue using DES whatever its shortcomings, convincing themselves that it is adequate for their needs. And DES, with its glaring vulnerabilities, will go on pretending to protect information for decades to come.
In privacy and computer security, real information is too hard to find. Most people don’t know what’s really going on, and many people who do know aren’t telling.

This book was written to reveal a hidden truth. The standard way that the US Government recommends that we make information secure and private, the “Data Encryption Standard” or DES, does not actually make that information secure or private. The government knows fairly simple ways to reveal the hidden information (called “cracking” or “breaking” DES).

Many scientists and engineers have known or suspected this for years. The ones who know exactly what the government is doing have been unable to tell the public, fearing prosecution for revealing “classified” information. Those who are only guessing have been reluctant to publish their guesses, for fear that they have guessed wrong.

This book describes a machine which we actually built to crack DES. The machine exists, and its existence can easily be verified. You can buy one yourself, in the United States; or can build one yourself if you desire. The machine was designed and built in the private sector, so it is not classified. We have donated our design to the public domain, so it is not proprietary. There is no longer any question that it can be built or has been built. We have published its details so that other scientists and engineers can review, reproduce, and build on our work. There can be no more doubt. DES is not secure.


**Chapters**

The first section of the book describes the Electronic Frontier Foundation's research project to build a machine to crack DES. The next section provides full technical details on the machine that we designed: for review, critique, exploration, and further evolution by the cryptographic research community. The final section includes several hard-to-find technical reports on brute force methods of cracking DES.

**Technical description**

Chapter 1, *Overview*, introduces our project and gives the basic architecture of the Electronic Frontier Foundation's DES-cracking machine.

Chapter 2, *Design Specification*, by Paul Kocher of Cryptography Research, provides specifications for the machine from a software author's point of view.

Chapter 3, *Hardware Specification*, by Advanced Wireless Technologies, provides specifications for the custom gate array chips, and the boards that carry them, from a hardware designer's point of view.

**Technical design details**

Chapter 4, *Scanning the Source Code*, explains how you can feed this book through an optical scanner and regenerate the exact source code needed to build the software and the specialized gate array chip that we designed.

Chapter 5, *Software Source Code*, contains a complete listing of the C-language software that runs on a PC and controls the DES-Cracker.

Chapter 6, *Chip Source Code*, contains a complete listing of the chip design language (VHDL) code that specifies how we designed the custom gate array chip.

Chapter 7, *Chip Simulator Source Code*, contains a complete listing of the C-language software that simulates the operation of the chip, for understanding how the chip works, and for generating test-vectors to make sure that the chips are properly fabricated.

Chapter 8, *Hardware Board Schematics*, provides schematic diagrams of the boards which provide power and a computer interface to the custom chips, as well as information on the layout of the boards and the backplanes that connect them.
Related Research Papers

Chapter 9, *Breaking One Million DES Keys* by Yvo Desmedt, is a 1987 paper proposing an interesting design for a machine that could search for many DES keys simultaneously.

Chapter 10, *Architectural considerations for cryptanalytic hardware*, by Ian Goldberg and David Wagner, is a 1996 study that explores cracking DES and related ciphers by using field-programmable gate array chips.

Chapter 11, *Efficient DES Key Search - An Update*, by Michael J. Wiener, revises for 1998 the technology estimates from his seminal 1993 paper, which was the first to include full schematic diagrams of a custom chip designed to crack DES.

Chapter 12, *About the Authors*, describes the foundation and the companies which collaborated to build this project.
Overview

Politics of Decryption

We began the Electronic Frontier Foundation’s DES Cracker project because of our interest in the politics of decryption.* The vulnerability of widely used encryption standards like DES is important for the public to understand.

A “DES Cracker” is a machine that can read information encrypted with the Data Encryption Standard (DES), by finding the key that was used to encrypt it. “Cracking DES” is a name for this search process. It is most simply done by trying every possible key until the right one is found, a tedious process called “brute-force search”.

If DES-encrypted information can easily be decrypted by those who are not intended to see it, the privacy and security of our infrastructures that use DES are at risk. Many political, social, and technological decisions depend on just how hard it is to crack DES.

We noticed an increasing number of situations in which highly talented and respected people from the U.S. Government were making statements about how long it takes to crack DES. In all cases, these statements were at odds with our own estimates and those of the cryptographic research community. A less polite way to say it is that these government officials were lying, incompetent, or both. They were stating that cracking DES is much more expensive and time-consuming than we believed it to be. A very credible research paper had predicted that a

*DES, the Data Encryption Standard, encrypts a confidential message into scrambled output under the control of a secret key. The input message is also known as “plaintext”, and the resulting output as “ciphertext”. The idea is that only recipients who know the secret key can decrypt the ciphertext to obtain the original message. DES uses a 56-bit key, so there are \(2^{56}\) possible keys.
machine could be built for $1.5 million, including development costs, that would crack DES in 3-1/2 hours. Yet we were hearing estimates of thousands of computers and weeks to years to crack a single message.

On Thursday, June 26, 1997 the U.S. House of Representatives' Committee on International Relations heard closed, classified testimony on encryption policy issues. The Committee was considering a bill to eliminate export controls on cryptography. After hearing this testimony, the Committee gutted the bill and inserted a substitute intended to have the opposite effect. A month later, a censored transcript of the hearing was provided; see http://jya.com/hir-hear.htm. Here are excerpts:

**Statement of Louis J. Freeh, Director, Federal Bureau of Investigation**

... And we do not have the computers, we do not have the technology to get either real-time access to that information or any kind of timely access.

If we hooked together thousands of computers and worked together over 4 months we might, as was recently demonstrated decrypt one message bit. That is not going to make a difference in a kidnapping case, it is not going to make a difference in a national security case. We don’t have the technology or the brute force capability to get to this information.

**Statement of William P. Crowell, Deputy Director, National Security Agency**

... I would go further and say there have been people who have said that Louis Freeh’s organization should just get smarter technically, and if they were just smarter technically, they would be able to break all of this stuff. I would like to leave you with just one set of statistics, and then I think I am going to close with just a few comments on the bill itself.

There is no brute force solution for law enforcement. [blacked out] A group of students -- not students -- the Internet gang last week broke a single message using 56-bit DES. It took 78,000 computers 96 days to break one message, and the headline was, DES has weak encryption.

He doesn’t consider that very weak. If that had been 64-bit encryption, which is available for export today, and is available freely for domestic use, that same effort would have taken 7,000 years. And if it had been 128-bit cryptography, which is what PGP is, pretty good privacy, it would have taken 8.6 trillion times the age of the universe.
Comments made later in the hearing

Chairman Gilman. Would you need added manpower resource and equipment if there is a need to decrypt? And would that add to your already difficult case of language translation in many of your wiretaps?

Director Freeh. We would certainly need those resources, but I think more importantly is the point that was made here. Contrary to the National Research Council recommendation that the FBI buy more computers and Bill Gates’ suggestion to me that we upgrade our research and development [blacked out---- ------------------------] American industry cannot do it, and that is decrypt real time encryption over a very minimal level of robustness. [blacked out-------] If you gave me $3 million to buy a Cray computer, it would take me how many years to do one message bit?

Mr. Crowell. 64 bits, 7,000 years.

Director Freeh. I don’t have that time in a kidnapping case. It would kill us.

On March 17, 1998, Robert S. Litt, Principal Associate Deputy Attorney General, testified to the U.S. Senate Judiciary Committee, Subcommittee on the Constitution, Federalism, and Property. The subject of the hearing was “Privacy in a Digital Age: Encryption and Mandatory Access”. Mr. Litt’s whole statement is available at http://www.computerprivacy.org/archive/03171998-4.shtml. The part relevant to DES cracking is:

Some people have suggested that this is a mere resource problem for law enforcement. They believe that law enforcement agencies should simply focus their resources on cracking strong encryption codes, using high-speed computers to try every possible key when we need lawful access to the plaintext of data or communications that is evidence of a crime. But that idea is simply unworkable, because this kind of brute force decryption takes too long to be useful to protect the public safety. For example, decrypting one single message that had been encrypted with a 56-bit key took 14,000 Pentium-level computers over four months; obviously, these kinds of resources are not available to the FBI, let alone the Jefferson City Police Department.

What’s Wrong With Their Statements?

Some of the testimony quoted may have been literally true; nevertheless, it is deceptive. All of the time estimates presented by Administration officials were based on use of general-purpose computers to do the job. But that’s fundamentally the wrong way to do it, and they know it.

A ordinary computer is ill-suited for use as a DES Cracker. In the first place, the design of DES is such that it is inherently very slow in software, but fast in hardware. Second, current computers do very little in parallel; the designers don’t know exactly what instructions will be executed, and must allow for all combinations.
The right way to crack DES is with special-purpose hardware. A custom-designed chip, even with a slow clock, can easily outperform even the fastest general-purpose computer. Besides, you can get many such chips on a single board, rather than the one or two on a typical computer's motherboard.

There are practical limits to the key sizes which can be cracked by brute-force searching, but since NSA deliberately limited the key size of DES to 56 bits, back in the 1970's when it was designed, DES is crackable by brute force. Today's technology might not be able to crack other ciphers with 64-bit or 128-bit keys — or it might. Nobody will know until they have tried, and published the details for scientific scrutiny. Most such ciphers have very different internal structure than DES, and it may be possible to eliminate large numbers of possible keys by taking advantage of the structure of the cipher. Some senior cryptographers estimated what key sizes were needed for safety in a 1996 paper;* they suggest that to protect against brute force cracking, today's keys should have a minimum of 75 bits, and to protect information for twenty years, a minimum of 90 bits.

The cost of brute-force searching also overstates the cost of recovering encrypted text in the real world. A key report on the real impact of encryption on law enforcement† reveals that there are no cases in which a lack of police access to encrypted files resulted in a suspected criminal going free. In most cases the plaintext was recovered by other means, such as asking the suspect for the key, or finding another copy of the information on the disk. Even when brute force is the method of choice, keys are seldom truly random, and can be searched in the most likely order.

**Export Controls and DES**

The U.S. Government currently restricts the ability of companies, individuals, and researchers to export hardware or software that includes the use of DES for confidentiality. These “export controls” have been a severe impediment to the development of security and privacy for networked computers, cellular phones, and other popular communications devices. The use of encryption algorithms stronger than DES is also restricted.

In December 1996, the government formally offered exporters the ability to incorporate DES, but nothing stronger, into their products. The catch is that these companies would have to sign an agreement with the government, obligating them to

---


install "key recovery" into their products within two years. Key recovery technology provides a way for the government to decrypt messages at will, by offering the government a copy of the key used in each message, in a way that the product's user cannot circumvent or control. In short, the government's offer was: collude with us to violate your customers' privacy, or we won't let you export any kind of secure products.

At the same time, the FBI was let into the group that reviews each individual company's application to export a cryptographic product. All reports indicate that the FBI is making good on the threat, by objecting to the export of all kinds of products that pose no threat at all to the national security (having been exportable in previous years before the FBI gained a voice). The FBI appears to think that by making itself hated and feared, it will encourage companies to follow orders. Instead it is encouraging companies to overturn the regulatory scheme that lets the FBI abuse the power to control exports. Industry started a major lobbying group called Americans for Computer Privacy (http://www.computerprivacy.org), which is attempting to change the laws to completely decontrol non-military encryption exports.

Some dozens of companies to signed up for key recovery, though it is unclear how many actually plan to follow through on their promise to deploy the technology. You will not find many of these companies trumpeting key recovery in their product advertisements. Users are wary of it since they know it means compromised security. If customers won't buy such products, companies know it makes no sense to develop them.

The best course for companies is probably to develop products that provide actual security, in some jurisdiction in the world which does not restrict their export. Some companies are doing so. The government's "compromise" offer discourages hesitant companies from taking this step, by providing a more moderate and conciliatory step that they can take instead. Companies that go to the effort to build overseas cryptographic expertise all use stronger technology than DES, as a selling point and to guard against early obsolescence. If those companies can be convinced to stay in the US, play the government's key-recovery game, and stick with DES, the government continues to win, and the privacy of the public continues to lose.

The success or failure of the government's carrot-and-stick approach depends on keeping industry and the public misled about DES's security. If DES-based products were perceived as insecure, there would be little reason for companies to sign away their customers' privacy birthrights in return for a mess of DES pottage. If DES-based products are perceived as secure, but the government actually knows that the products are insecure, then the government gets concessions from compa-
nies, without impacting its ability to intercept communications. Keeping the public ignorant gives the government the best of both worlds.

**Political Motivations and EFF’s Response**

We speculate that government officials are deliberately misleading the public about the strength of DES encryption:

- To encourage the public to continue using DES, so their agencies can eavesdrop on the public.
- To prevent the widespread adoption of stronger standards than DES, which the government would have more trouble decrypting.
- To offer DES exportability as a bargaining-chip, which actually costs the government little, but is perceived to be valuable.
- To encourage policy-makers such as Congressmen or the President to impose drastic measures such as key recovery, in the belief that law enforcement has a major encrypted-data problem and no practical way to crack codes.

As advocates on cryptography policy, we found ourselves in a hard situation. It appeared that highly credible people were either deliberately lying to Congress and to the public in order to advance their own harmful agendas, or were advocating serious infringement of civil liberties based on their own ignorance of the underlying issues. Most troubling is the possibility that they were lying. Perhaps these government executives merely saw themselves as shielding valuable classified efforts from disclosure. As advocates of good government, we do not see that classifying a program is any justification for an official to perjure themselves when testifying about it. (Declining to state an opinion is one thing; making untruthful statements as if they were facts is quite another.)

The National Research Council studied encryption issues and published a very complete 1996 report.* The most interesting conclusion of their report was that “the debate over national cryptography policy can be carried out in a reasonable manner on an unclassified basis”. This presumes good faith on the part of the agencies who hide behind classified curtains, though. If it turns out that their public statements are manipulative falsehoods, an honest and reasonable public debate must necessarily exclude them, as dishonest and unreasonable participants.

In the alternative, if poor policy decisions are being made based on the ignorance or incompetence of senior government officials, the role of honest advocates should be to inform the debate.

In response to these concerns, EFF began a research program. Our research results prove that DES can be cracked quickly on a low budget. This proves that these officials were either lying or incompetent. The book you are holding documents the research, and allows it to be validated by other scientists.

**Goals**

The goal of EFF's DES Cracker research project is to determine just how cheap or expensive it is to build a machine that cracks DES usefully.

Technically, we were also interested in exploring good designs for plaintext recognizers. These are circuits that can notice when the result of decryption is likely enough to be correct that specialized software — or a human — should look at it. Little research has been published on them,* yet they are a vital part of any efficient system for cryptanalysis.

Merely doing the research would let EFF learn the truth about the expense of cracking DES. But only publishing the research and demonstrating the machine would educate the public on the truth about the strength of DES. Press releases and even technical papers would not suffice; the appearance of schematics for a million-dollar DES Cracker in Michael Wiener's excellent 1993 paper should have been enough. But people still deploy DES, and Congressmen blindly accept the assurances of high officials about its strength.

There are many people who will not believe a truth until they can see it with their own eyes. Showing them a physical machine that can crack DES in a few days is the only way to convince some people that they really cannot trust their security to DES.

Another set of people might not believe our claims unless several other teams have reproduced them. (This is a basic part of the scientific method.) And many people will naturally be interested in how such a box works, and how it was built for only about $200,000. This book was written for such people. It contains the complete specifications and design documents for the DES Cracker, as well as circuit diagrams for its boards, and complete listings of its software and its gate array design. The full publication of our design should enable other teams to rapidly reproduce, validate, and improve on our design.

---

History of DES Cracking

DES Crackers have been mentioned in the scientific and popular literature since the 1970's. Whitfield Diffie's Foreword describes several of them. The most recent detailed description was in a paper by Michael Wiener of Bell Northern Research in 1993. Wiener's paper included a detailed hardware design of a DES Cracker built with custom chips. The chips were to be built into boards, and the boards into mechanical "frames" like those of telephone central office switches. A completed design would have cost about a million dollars and would determine a DES key from known plaintext and known ciphertext in an average of 3-1/2 hours (7 hours in the worst case).

Mr. Wiener updated his conclusions in 1998, adjusting for five years of technological change. His update paper is included in this book, thanks to the courtesy of RSA Data Security, which originally published his update.

Ian Goldberg and David Wagner of the University of California at Berkeley took a different approach. Their design used a "field programmable gate array" (FPGA), which is a chip that can be reprogrammed after manufacturing into a variety of different circuits.

FPGA chips are slower than the custom chips used in the Wiener design, but can be bought quickly in small quantities, without a large initial investment in design. Rather than spend a big chunk of a million dollars to design a big machine, these researchers bought one or two general purpose chips and programmed them to be a slow DES Cracker. This let them quickly measure how many slow chips they would need to pile up to make a practical DES Cracker. Their paper is also included in this book.

EFF's DES Cracker Project

The Electronic Frontier Foundation began its investigation into DES Cracking in 1997. The original plan was to see if a DES Cracker could be built out of a machine containing a large number of FPGA's.

Large machines built out of FPGAs exist in the commercial market for use in simulating large new chip designs before the chip is built. A collection of thousands of relatively incapable FPGA chips can be put together to simulate one very capable custom chip, although at 1/10th or 1/100th of the speed that the eventual custom chip would run at. This capability is used by chip designers to work the "bugs" out of their chip before committing to the expensive and time-consuming step of fabricating physical chips from their design.

EFF never got access to such a chip simulator. Instead, our investigations led us to Paul Kocher of Cryptography Research. Paul had previously worked with a team
of hardware designers who knew how to build custom gate array chips cheaply, in batches of a few thousand chips at a time.

Paul and EFF met with the chip designers at Advanced Wireless Technologies, and determined that a workable DES Cracker could be built on a budget of about $200,000. The resulting machine would take less than a week, on average, to determine the key from a single 8-byte sample of known plaintext and ciphertext. Moreover, it would determine the key from a 16-byte sample of ciphertext in almost the same amount of time, if the statistical characteristics of the plaintext were known or guessable. For example, if the plaintext was known to be an electronic mail message, it could find all keys that produce plaintext containing nothing but letters, numbers, and punctuation. This makes the machine much more usable for solving real-world decryption problems.

There is nothing revolutionary in our DES Cracker. It uses ordinary ideas about how to crack DES that have been floating around in the cryptographic research community for many years. The only difference is that we actually built it, instead of just writing papers about it. Very similar machines could have been built last year, or the year before, or five or ten years ago; they would have just been slower or more expensive.

**Architecture**

The design of the EFF DES Cracker is simple in concept. It consists of an ordinary personal computer connected with a large array of custom chips. Software in the personal computer instructs the custom chips to begin searching, and interacts with the user. The chips run without further help from the software until they find a potentially interesting key, or need to be directed to search a new part of the key space. The software periodically polls the chips to find any potentially interesting keys that they have turned up.

The hardware's job isn't to find the answer, but rather to eliminate most of the answers that are incorrect. Software is then fast enough to search the remaining potentially-correct keys, winnowing the "false positives" from the real answer. The strength of the machine is that it replicates a simple but useful search circuit thousands of times, allowing the software to find the answer by searching only a tiny fraction of the key space.

As long as there is a small bit of software to coordinate the effort, the problem of searching for a DES key is "highly parallelizable". This means the problem can be usefully solved by many machines working in parallel, simultaneously. For example, a single DES-Cracker chip could find a key by searching for many years. A thousand DES-Cracker chips can solve the same problem in one thousandth of the time. A million DES-Cracker chips could theoretically solve the same problem in
about a millionth of the time, though the overhead of starting each chip would become visible in the time required. The actual machine we built contains 1536 chips.

When conducting a brute-force search, the obvious thing to do is to try every possible key, but there are some subtleties. You can try the keys in any order. If you think the key isn’t randomly selected, start with likely ones. When you finally find the right key, you can stop; you don’t have to try all the rest of the keys. You might find it in the first million tries; you might find it in the last million tries. On average, you find it halfway through (after trying half the keys). As a result, the timings for brute-force searches are generally given as the average time to find a key. The maximum time is double the average time.

**Search units**

The search unit is the heart of the EFF DES Cracker; it contains thousands of them.

A search unit is a small piece of hardware that takes a key and two 64-bit blocks of ciphertext. It decrypts a block of ciphertext with the key, and checks to see if the resulting block of plaintext is “interesting”. If not, it adds 1 to the key and repeats, searching its way through the key space.

If the first decryption produces an “interesting” result, the same key is used to decrypt the second block of ciphertext. If both are interesting, the search unit stops and tells the software that it has found an interesting key. If the second block’s decryption is uninteresting, the search unit adds one to the key and goes on searching the key space.

When a search unit stops after finding an interesting result, software on the host computer must examine the result, and determine whether it’s the real answer, or just a “false positive”. A false positive is a plaintext that looked interesting to the hardware, but which actually isn’t a solution to the problem. The hardware is designed to produce some proportion of false positives along with the real solution. (The job of the hardware isn’t to find the answer, but to eliminate the vast majority of the non-answers.) As long as the false positives don’t occur so rapidly that they overwhelm the software’s ability to check and reject them, they don’t hurt, and they simplify the hardware and allow it to be more general-purpose. For the kinds of problems that we’re trying to solve, the hardware is designed to waste less than 1% of the search time on false positives.


**Recognizing interesting plaintext**

What defines an interesting result? If we already know the plaintext, and are just looking for the key, an interesting result would be if the plaintext from this key matches our known block of plaintext. If we don’t know the plaintext, perhaps the guess that it’s all composed of letters, digits, and punctuation defines “interesting”. The test has to be simple yet flexible. We ended up with one that’s simple for the hardware, but a bit more complicated for the software.

Each result contains eight 8-bit bytes. First, the search unit looks at each byte of the result. Such a byte can have any one of 256 values. The search unit is set up with a table that defines which of these 256 byte values are “interesting” and which are uninteresting. For example, if the plaintext is known to be all numeric, the software sets up the table so that the ten digits (0 to 9) are interesting, and all other potential values are uninteresting.

The result of decrypting with the wrong key will look pretty close to random. So the chance of having a single byte look “interesting” will be based on what fraction of the 256 values are defined to be “interesting”. If, say, 69 characters are interesting (A-Z, a-z, 0-9, space, and a few punctuation characters), then the chance of a random byte appearing to be interesting is 69/256 or about 1/4. These don’t look like very good odds; the chip would be stopping on one out of every four keys, to tell the software about “interesting” but wrong keys.

But the “interest” test is repeated on each byte in the result. If the chance of having a wrong key’s byte appear interesting is 1/4, then the chance of two bytes appearing interesting is 1/4 of 1/4, or 1/16th. For three bytes, 1/4th of 1/4th of 1/4th, or 1/64th. By the time the chip examines all 8 bytes of a result, it only makes a mistake on 1/65536th of the keys (1/4^8 keys).

That seems like a pretty small number, but when you’re searching through 72,057,594,037,927,936 keys (2^56 keys, or 72 quadrillion keys), you need all the help you can get. Even having the software examine 1/65536th of the possible keys would require looking at 1,099,511,627,776 keys (2^40 or about a trillion keys). So the chip provides a bit more help.

This help comes from that second block of ciphertext. If every byte of a result looks interesting when the first block of ciphertext is decrypted, the chip goes back around and decrypts the second block of ciphertext with the same key. This divides the “error rate” by another factor of 65536, leaving the software with only 16,777,216 (2^24 or about sixteen million) keys to look at. Software on modern computers is capable of handling this in a reasonable amount of time.

(If we only know one block of ciphertext, we just give the chip two copies of the same ciphertext. It will test both copies, and eventually tell us that the block is
interesting. The amount of time it spends checking this “second block” is always a tiny fraction of the total search time.)

In the plaintext recognizer there are also 8 bits that lets us specify which bytes of a plaintext are interesting to examine. For example, if we know or suspect the contents of the first six bytes of a plaintext value, but don’t know anything about the last two bytes, we can search for keys which match in just those six bytes.

**Known plaintext**

The chips will have many fewer “false positives” if the plaintext of the message is known, instead of just knowing its general characteristics. In that case, only a small number of byte values will be “interesting”. If the plaintext has no repeated byte values, only eight byte values will be interesting, instead of 69 as above.

For example, if the plaintext block is “hello th”, then only the six byte values “h”, “e”, “l”, “o”, space, and “t” are interesting. If a plaintext contains only these bytes, it is interesting. We’ll get some “false positives” since many plaintexts like “tholo tt” would appear “interesting” even though they don’t match exactly.

Using this definition of “interesting”, a byte resulting from a wrong key will look interesting only about 8/256ths of the time, or 1/32nd of the time. All eight bytes resulting from a wrong key will look interesting only 1/32nd to the eighth power (1/32nd of 1/32nd of 1/32nd of 1/32nd of 1/32nd of 1/32nd of 1/32nd of 1/32nd) of the time, or 1/1,099,511,627,776th of the time (1/2^40 of the time). In other words, a search unit can try an average of a trillion keys before reporting that a wrong key looks interesting. This lets it search for a long time without slowing down or bothering the software.

**Speed**

Once you get it going, a search unit can do one decryption in 16 clock cycles. The chips we have built can run with a clock of 40 Mhz (40 million cycles per second). Dividing 16 into 40 million shows that each search unit can try about 2.5 million keys per second.

In building the search units, we discovered that we could make them run faster if we used simpler circuitry for adding 1 to a key. Rather than being able to count from a key of 0 all the way up to a key of all ones, we limited the adder so that it can only count the bottom 32 bits of the key. The top 24 bits always remain the same. At a rate of 2.5 million keys per second, it takes a search unit 1717 seconds (about half an hour) to search all the possible keys that have the same top 24 bits. At the end of half an hour, the software has to stop the chip, reload it with a new value in the top 24 bits, and start it going again.
Feedback Modes

The chip can also decrypt ciphertext that was encrypted in “Cipher Block Chaining” mode. In this mode, the ciphertext of each block is exclusive-OR'd into the plaintext of the next block before it is encrypted. (An “initialization vector” is exclusive-OR'd into the first block of plaintext.) The search unit knows how to exclusive-OR out an Initialization Vector (IV) after decrypting the first cyphertext, and to exclusive-OR out the first cyphertext after decrypting the second one. The software specifies the IV at the same time it provides the cyphertext values.

Blaze Challenge

In June, 1997 Matt Blaze, a cryptography researcher at AT&T, proposed a different sort of cryptographic challenge. He wanted a challenge that not even the proponent knew how to solve, without either doing a massive search of the key-space, or somehow cryptanalyzing the structure of DES.

His challenge is merely to find a key such that a ciphertext block of the form XXXXXXXX decrypts to a plaintext block of the form YYYYYYYY, where X and Y are any fixed 8-bit value that is repeated across each of the eight bytes of the block.

We added a small amount of hardware to the search units to help with solving this challenge. There is an option to exclusive-OR the right half of the plaintext into the left half, before looking to see if the plaintext is “interesting”. For plaintexts of the form YYYYYYYY, this will result in a left half of all zeros. We can then set up the plaintext recognizer so it only looks at the left half, and only thinks zeroes are interesting. This will produce a large number of false positives (any plaintext where the left and right halves are equal, like ABCDABCD), but software can screen them out with only about a 1% performance loss.

Structure Of The Machine

Now that you know how a single search unit works, let's put them together into the whole machine.

Each search unit fits inside a custom chip. In fact, 24 search units fit inside a single chip. All the search units inside a chip share the same ciphertext blocks, initialization vector, and the same plaintext-recognizer table of “interesting” result values. Each search unit has its own key, and each can be stopped and started independently.

The chip provides a simple interface on its wires. There are a few signals that say whether any of the search units are stopped, some address and data wires so that
the software can read and write to the search units, and wires for electrical power and grounding.

Since each search unit tries 2.5 million keys per second, a chip with 24 search units will try 60 million keys per second. But there are a lot of keys to look at. For a single chip, it would take 6,950 days (about 19 years) to find the average key, or 38 years to search the entire key space. Since we don't want to wait that long, we use more than one chip.

Each chip is mounted onto a large circuit board that contains 64 chips, along with a small bit of interface circuitry. The board blinks a light whenever the software is talking to that board. 64 other lights show when some search unit in each chip has stopped. In normal operation the software will talk to the board every few seconds, to check up on the chips. The chips should only stop every once in a while, and should be quickly restarted by the software.

The boards are designed to the mechanical specifications of “9U” VMEbus boards (about 15” by 15”). VMEbus is an industrial standard for computer boards, which was popular in the 1980s. We used the VMEbus form factor because it was easy to buy equipment that such boards plug into; we don't actually use the VMEbus electrical specifications.

9U VMEbus boards are much larger than the average interface card that plugs into a generic PC, so a lot more chips can be put onto them. Also, 9U VMEbus boards are designed to supply a lot of power, and our DES Cracker chips need it.

Since each chip searches 60 million keys per second, a board containing 64 chips will search 3.8 billion keys per second. Searching half the key space would take the board about 109 days. Since we don't want to wait that long either, we use more than one board.

The boards are mounted into chassis, also called “card cages”. In the current design, these chassis are recycled Sun workstation packages from about 1990. Sun Microsystems built a large number of systems that used the large 9U VMEbus boards, and provide excellent power and cooling for the boards. The Sun-4/470 chassis provides twelve slots for VMEbus boards, and can easily be modified to handle our requirements. Subsequent models may use other physical packaging.

Each chassis has a connector for a pair of “ribbon cables” to connect it to the next chassis and to the generic PC that runs the software. The last chassis will contain a “terminator”, rather than a connection to the next chassis, to keep the signals on the ribbon cable from getting distorted when they reach the end of the line.

Since each board searches 3.8 billion keys per second, a chassis containing 12 boards will search 46 billion keys per second. At that rate, searching half the key space takes about 9 days. One chassis full of boards is about 25% faster than the
entire worldwide network of machines that solved the RSA “DES-II” challenge in February 1998, which was testing about 34 billion keys per second at its peak.

Since an informal design goal for our initial DES Cracker was to crack an average DES key in less than a week, we need more than 12 boards. To give ourselves a comfortable margin, we are using 24 boards, which we can fit into two chassis. They will search 92 billion keys per second, covering half the key space in about 4.5 days. If the chips consume too much power or produce too much heat for two chassis to handle,* we can spread the 24 boards across three chassis.

Table 1-1: Summary of DES Cracker performance

<table>
<thead>
<tr>
<th>Device</th>
<th>How Many In Next Device</th>
<th>Keys/Sec</th>
<th>Days/avg search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Unit</td>
<td>24</td>
<td>2,500,000</td>
<td>166,800</td>
</tr>
<tr>
<td>Chip</td>
<td>64</td>
<td>60,000,000</td>
<td>6,950</td>
</tr>
<tr>
<td>Board</td>
<td>12</td>
<td>3,840,000,000</td>
<td>109</td>
</tr>
<tr>
<td>Chassis</td>
<td>2</td>
<td>46,080,000,000</td>
<td>9.05</td>
</tr>
<tr>
<td>EFF DES Cracker</td>
<td></td>
<td>92,160,000,000</td>
<td>4,524</td>
</tr>
</tbody>
</table>

We designed the search unit once. Then we got a speedup factor of more than 36,000 to 1 just by replicating it 24 times in each chip and making 1500 chips. This is what we meant by “highly parallelizable”.

**Budget**

The whole project was budgeted at about US$210,000. Of this, $80,000 is for the labor of designing, integrating, and testing the DES Cracker. The other $130,000 is for materials, including chips, boards, all other components on the boards, card cages, power supplies, cooling, and a PC.

The software for controlling the DES Cracker was written separately, as a volunteer project. It took two or three weeks of work.

The entire project was completed within about eighteen months. Much of that time was used for preliminary research, before deciding to use a custom chip rather than FPGA’s. The contract to build custom chips was signed in September, 1997, about eight months into the project. The team contained less than ten people, none of whom worked full-time on the project. They include a project manager, software designer, programmer, chip designer, board designer, hardware technicians, and hardware managers.

* At publication time, we have tested individual chips but have yet not built the full machine. If the chips’ power consumption or heat production is excessive in a machine containing 1500 chips, we also have the option to reduce the chips’ clock rate from 40 MHz down to, say, 30 MHz. This would significantly reduce the power and heat problems, at a cost of 33% more time per search (6 days on average).
We could have reduced the per-chip cost, or increased the chip density or search speed, had we been willing to spend more money on design. A more complex design could also have been flexible enough to crack other encryption algorithms. The real point is that for a budget that any government, most companies, and tens of thousands of individuals could afford, we built a usable DES Cracking machine. The publication of our design will probably in itself reduce the design cost of future machines, and the advance of semiconductor technology also makes this cost likely to drop. In five years some teenager may well build her own DES Cracker as a high school science fair project.

**Who Else Is Cracking DES?**

If a civil liberties group can build a DES Cracker for $200,000, it’s pretty likely that governments can do the same thing for under a million dollars. (That’s a joke.) Given the budget and mission of the US National Security Agency, they must have started building DES Crackers many years ago. We would guess that they are now on their fourth or fifth generation of such devices. They are probably using chips that are much faster than the ones we used; modern processor chips can run at more than 300 Mhz, eight times as fast as our 40 Mhz chips. They probably have small “field” units that fit into a suitcase and crack DES in well under a day; as well as massive central units buried under Ft. Meade, that find the average DES key in seconds, or find thousands of DES keys in parallel, examining thousands of independent intercepted messages.

Our design would scale up to finding a DES key in about half an hour, if you used 333,000 chips on more than 5,200 boards. The boards would probably require about 200 parallel port cards to communicate with them; an IBM-compatible PC could probably drive four such cards, thus requiring about 50 PC’s too. The software required would be pretty simple; the hard part would be the logistics of physical arrangement and repair. This is about 200 times as much hardware as the project we built. A ridiculously high upper bound on the price of such a system would be 200 times the current project price, or $40 million.

Of course, if we were going to build a system to crack DES in half an hour or less, using a third of a million chips, it would be better to go back to the drawing board and design from scratch. We’d use more modern chip fabrication processes; a higher-volume customer can demand this. We’d spend more on the initial design and the software, to produce a much cheaper and simpler total system, perhaps allowing boards full of denser, faster, lower-voltage chips to use a small onboard processor and plug directly into an Ethernet. We’d work hard to reduce the cost of each chip, since there would be so many of them. We’d think about how to crack multiple DES keys simultaneously.
It would be safe to assume that any large country has DES Cracking machines. After the publication of this book wakes them up, probably more small countries and some criminal organizations will make or buy a few DES Crackers. That was not the intent of the book; the intent was to inform and warn the targets of this surveillance, the builders of equipment, and the policy makers who grapple with encryption issues.

**What To Do If You Depend On DES**

Don’t design anything else that depends on single DES.

Take systems out of service that use permanently fixed single-DES keys, or superencrypt the traffic at a higher level. Superencryption requires special care, though, to avoid providing any predictable headers that can be used to crack the outer DES encryption.

Start changing your software and/or hardware to use a stronger algorithm than DES.

Three-key Triple-DES is an obvious choice, since it uses the same block size and can possibly use the same hardware; it just uses three keys and runs DES three times (encrypting each block with the first key, decrypting it with the second, then encrypting it with the third). The strength of Triple-DES is not known with any certainty, but it is certainly no weaker than single DES, and is probably substantially stronger. Beware of “mixed up” variants or modes of Triple-DES; research by Eli Biham* and David Wagner† shows that they are significantly weaker than the straightforward Triple-DES, and may be even weaker than single-DES. Use three copies of DES in Electronic Code Book (ECB) mode as a basic primitive. You can then build a mode such as Cipher Feedback mode using the primitive ECB 3DES.

The US Government is tardily going through a formal process to replace the DES. This effort, called the Advanced Encryption Standard, will take several years to decide on a final algorithm, and more years for it to be proven out in actual use, and carefully scrutinized by public cryptanalysts for hidden weaknesses. If you are designing products to appear five to ten years from now, the AES might be a good source of an encryption algorithm for you.

The reason that the AES is tardy is because the NSA is believed to have blocked previous attempts to begin the process over the last decade. In recent years NSA

---


has tried, without success, to get the technical community to use classified, NSA-designed encryption algorithms such as Skipjack, without letting the users subject these algorithms to public scrutiny. Only after this effort failed did they permit the National Institute of Standards and Technology to begin the AES standardization process.

**Conclusion**

The Data Encryption Standard has served the public pretty well since 1975. But it was designed in an era when computation cost real money, when massive computers hunkered on special raised flooring in air-conditioned inner sanctums. In an era when you can carry a supercomputer in your backpack, and access millions of machines across the Internet, the Data Encryption Standard is obsolete.

The Electronic Frontier Foundation hopes that this book inspires a new level of truth to enter the policy debates on encryption. In order to make wise choices for our society, we must make well-informed choices. Great deference has been paid to the perspective and experience of the National Security Agency and Federal Bureau of Investigation in these debates. This is particularly remarkable given the lack of any way for policy-makers or the public to check the accuracy of many of their statements.* (The public cannot even bear many of their statements, because they are classified as state secrets.) We hope that the crypto policy debate can move forward to a more successful and generally supported policy. Perhaps if these agencies will consider becoming more truthful, or policy-makers will stop believing unverified statements from them, the process can move more rapidly to such a conclusion.

---

* DES cracking is not the only issue on which agency credibility is questionable. For example, the true extent of the law enforcement problem posed by cryptography is another issue on which official dire predictions have been made, while more careful and unbiased studies have shown little or no impact. The validity of the agencies' opinion of the constitutionality of their own regulations is also in doubt, having been rejected two decades ago by the Justice Department, and declared unconstitutional in 1997 by a Federal District Court. The prevalence of illegal wiretapping and communications interception by government employees is also in question; see for example the Los Angeles Times story of April 26, 1998, "Can the L.A. Criminal-Justice System Work Without Trust?"
In This chapter:
• On-Chip Registers
• Commands
• Search Unit Operation
• Sample Programming Descriptions
• Scalability and Performance
• Host Computer Software
• Glossary

Design for DES
Key Search Array

Cryptography Research
and
Advanced Wireless Technologies, Inc.

On-Chip Registers
Each chip contains the following registers. They are addressed as specified in Figure 2-1.

Ciphertext0 (64 bits = 8 bytes)
The value of the first ciphertext being searched. Ciphertext0 is identical in all search units and is set only once (when the search system is first initialized).

Ciphertext1 (64 bits = 8 bytes)
The value of the second ciphertext being searched. Ciphertext1 is identical in all search units and is set only once (when the search system is first initialized).

PlaintextByteMask (8 bits)
The plaintext byte selector. One-bits in this register indicate plaintext bytes that should be ignored when deciding whether or not the plaintext produced by a particular key is possibly correct. This mask is helpful when only a portion of the plaintext's value is known. For example, if the first 5 bytes equal a known header but the remaining three are unknown, a PlaintextByteMask of 0x07 would be used.

PlaintextXorMask (64 bits = 8 bytes)
This register is XORed with decryption of ciphertext0. This is normally filled with
the CBC mode IV.

PlaintextVector (256 bits = 8 bytes)
Identifies allowable plaintext byte values (ignoring those masked by the PlaintextByteMask). If, for any plaintext byte $P[i=0..7]$, bit $P[i]$ is not set, the decryption key will be rejected. PlaintextVector is identical in all search units and is set only once (when the search system is first initialized).

SearchInfo (8 bits)
The bits in SearchInfo describe how the correct plaintext identification function works. Bits of SearchInfo are defined as follows:
bit 0 = UseCBC
If this bit is set, Ciphertext0 is XORed onto the plaintext produced by decrypting Ciphertext1 before the plaintext is checked. This bit is used when checking CBC-mode ciphertexts.

bit 1 = ExtraXOR
If set, the right half of the resulting plaintext is XORed onto the left before any plaintext checking is done. ExtraXOR and UseCBC cannot be used together.

bit 2 = ChipAllActive
If cleared, one or more search units in this chip have halted (e.g., SearchActive is zero). This value is computed by ANDing the SearchActive bits of all search units' SearchStatus bytes. The inverse of this value is sent out on a dedicated pin, for use in driving a status LED which lights up whenever the chip halts.

bit 3 = BoardAllActive
This pin is the AND of the ChipAllActive lines of this chip and all later chips on the board. This is implemented by having each chip n take in chip n+1’s BoardAllActive line, AND it with its own ChipAllActive line, and output the result to chip n-1 for its BoardAllActive computation. This makes it possible to find which chip on a board has halted by querying log₂N chips, where N is the number of chips on the board. If BoardAllActiveEnable is not set to 1, BoardAllActive simply equals the BoardAllActiveInput pin, regardless of the chip's internal state.

bit 4 = BoardAllActiveEnable
If this value is set to 0 then BoardAllActive always equals the BoardAllActiveInput pin, regardless of whether all search units on the board are active. If this bit is set to 1, then the BoardAllActive register (and output) are set to reflect the internal state of the chip ANDed with the input pin.

bits 5-7 = Unused

KeyCounter (56 bits)
The value of the key currently being checked. The KeyCounter is updated very frequently (i.e., once per key tested). A unique KeyCounter value is assigned to every search unit. When the search unit halts after a match, KeyCounter has already been incremented to the next key; the match was on the previous key.

SearchCommandAndStatus (8 bits)
The bits in SearchStatus describe the current search state of a specific search unit. A unique SearchStatus register is allocated for each search unit. Bits of SearchStatus are allocated as follows:
bit 0 = SearchActive
Indicates whether the search is currently halted (0=halted, 1=active). The computer sets this bit to begin a search, and it is cleared by the search unit if a matching candidate key is found. The host computer checks the status of this bit periodically and, if it is zero, reads out the key then restarts the search. (See also ChipAllActive and BoardAllActive in the SearchInfo register.)

bit 1 = CiphertextSelector
Indicates whether the search engine is currently checking Ciphertext0 or Ciphertext1. (0=Ciphertext0, 1=Ciphertext1). If this bit is clear, the search engine decrypts Ciphertext0 and either sets CiphertextSelector to 1 (if the plaintext passes the checks) or increments KeyCounter (if the plaintext does not pass). If this bit is set, the search engine decrypts Ciphertext1 and either sets SearchActive to 0 (if the plaintext passes the checks) or sets CiphertextSelector to 0 and increments KeyCounter (if the plaintext does not pass).

bits 2-7 = Unused

Commands
In order to be able to address each search unit separately, each can be addressed uniquely by the combination of its location on the chip, the location of the chip on the board, and board's identifier. The BoardID is interpreted off-chip; each chip has a board select pin, which notifies the chip when the board has been selected. Chip ID matching is done inside each ASIC; the ID pins of the ASIC are wired to the chip's ID.

All commands are originated by the computer go via a bus which carries 8 bits for BoardID/ChipID/Register address, 8 bits for data, and a few additional bits for controls.

To do a search, the host computer will program the search units as shown in Figure 2-2. (N is the total number of search units, numbered from 0 to N-1, each with a unique BoardID/ChipID/Register address.)

Search Unit Operation
Each search unit contains a DES engine, which performs DES on two 32-bit registers L/R using the key value in KeyCounter. Each search unit goes through the process detailed in Figure 2-3, and never needs to halt. If registers are updated during the middle of this process, the output is meaningless (which is fine, since an incorrect output is statistically almost certain to not be a match).
Figure 2-2: Example algorithm for programming the search array using host computer

This is a very simple algorithm intended only as an example. The actual software will use more intelligent search techniques, using the BoardAllActive and ChipAllActive lines.

Load Ciphertext0, Ciphertext1, PlaintextXorMask, PlaintextByteMask, PlaintextVector, and SearchInfo into each chip.

For i = 0 upto N-1
    Set SearchStatus in search unit i to 0 while loading the key.
    Set KeyCounter of search unit i to ((256)(i) / N).
    Set SearchStatus in search unit i to 1 to enable SearchActive.
EndFor

While correct key has not been found:
    For i = 0 upto N-1:
        Read SearchStatus from search unit i.
        Check SearchActive bit.
        If SearchActive is set to 0:
            Read KeyCounter from search unit i.
            Subtract 1 from the low 32 bits of the key.
            Perform a DES operation at the local computer to check the key.
            If the key is correct, the search is done.
            Set the SearchActive bit of SearchStatus to restart the search.
        EndIf
    EndFor
EndWhile

Sample Programming Descriptions

This section describes how the system will be programmed for some typical operations.

Known ciphertext/plaintext (ECB, CBC, etc.)

If a complete ciphertext/plaintext block is known, this mode is used. This works for most DES modes (ECB, CBC, counter, etc.), but does require a full plaintext/ciphertext pair.

PlaintextVector

For this search, there are 8 (or fewer) unique plaintext bytes in the known plaintext. The bits corresponding to these bytes are set in PlaintextVector, but all other bits are set to 0.
Figure 2-3: Search unit operation

1. If CiphertextSelector is 0, then Let L/R = Ciphertext0. If CiphertextSelector is 1, then Let L/R = Ciphertext1.

2. Decrypt L/R using the key in KeyCounter, producing a candidate plaintext in L/R.

3. If ExtraXOR is 1, then Let L = L XOR R. If CiphertextSelector is 0, then Let L/R = L/R XOR PlaintextXorMask. If CiphertextSelector is 1 and UseCBC is 1, then: Let L/R = L/R XOR Ciphertext0.

4. If SearchActive = 1 AND (PlaintextByteMask[0x80] = 0 AND PlaintextVector[byte 0 of L] is 0) OR (PlaintextByteMask[0x40] = 0 AND PlaintextVector[byte 1 of L] is 0) OR (PlaintextByteMask[0x20] = 0 AND PlaintextVector[byte 2 of L] is 0) OR (PlaintextByteMask[0x10] = 0 AND PlaintextVector[byte 3 of L] is 0) OR (PlaintextByteMask[0x08] = 0 AND PlaintextVector[byte 0 of R] is 0) OR (PlaintextByteMask[0x04] = 0 AND PlaintextVector[byte 1 of R] is 0) OR (PlaintextByteMask[0x02] = 0 AND PlaintextVector[byte 2 of R] is 0) OR (PlaintextByteMask[0x01] = 0 AND PlaintextVector[byte 3 of R] is 0)) then: Let CiphertextSelector = 0. Increment KeyCounter.
   else
   If CiphertextSelector is 1 then Let SearchActive = 0.
   Let CiphertextSelector = 1.

5. Go to step 1.

Ciphertext0
Equals the ciphertext block.

Ciphertext1
Equals the ciphertext block.

SearchInfo
UseCBC and ExtraXOR are both set to 0.

PlaintextByteMask
Set to 0x00 (all bytes used).
PlaintextXorMask
Set to 0x0000000000000000.

Because the plaintext byte order does not matter, there are 8 acceptable values for each ciphertext byte, or \(8^8 = 2^{24} = 16.7\) million possible ciphertexts which will satisfy the search criteria. The probability that an incorrect ciphertext will pass is \(2^{24} / 2^{64}\), so over a search of \(2^{55}\) keys there will be an average of \((2^{55})(2^{24} / 2^{64})\), or 32768 false positives which will need to be rejected by the controlling computer. Because the Ciphertext0 and Ciphertext1 selections are identical, any false positives that pass the first test will also pass the second test. (The performance penalty is negligible; the search system will do two DES operations on each of the 32768 false positive keys, but only one DES operation on all other incorrect keys.)

**ASCII text (ECB or CBC)**

A minimum of two adjacent ciphertexts (16 bytes total) are required for ASCII-only attacks.

PlaintextVector
Set only the bits containing acceptable ASCII characters. For normal text, this would normally include 55 of the 256 possible characters occur (10=\(\text{line feed}\), 13=\(\text{carriage return}\), 32=\(\text{space}\), 65-90=\(\text{capital letters}\), and 97-122=\(\text{lowercase letters}\)).

Ciphertext0
Equals the first ciphertext.

Ciphertext1
Equals the second ciphertext.

SearchInfo
UseCBC is set to 0 if ECB, or set to 1 if the ciphertext was produced using CBC. ExtraXOR is set to 0.

PlaintextByteMask
Set to 0x00 (all bytes used).

PlaintextXorMask
Set to 0x0000000000000000 for ECB, to IV for CBC.

The probability that the two (random) candidate plaintexts produced by an incorrect key will contain only the ASCII text characters listed above is \((55/256)^{16}\). In a search, there will thus be an average of \(2^{55} (55/256)^{16} = 742358\) false positives which need to be rejected by the computer. For one key in about 220,000, the first check will pass and an extra DES will be required. (The time for these extra DES operations is insignificant.) Idle time lost while waiting for false positives to be cleared is also insignificant. If the computer checks each search unit’s SearchActive flag once per second, a total of 0.5 search unit seconds will be wasted for every
false positive, or a total of 103 search-unit hours, out of about 4 million search-unit hours for the whole search.

When programming CBC mode, note that the PlaintextXorMask must be set to the IV (or the previous ciphertext, if the ciphertext being attacked is not in the first block).

**Matt Blaze’s Challenge**

The goal is to find a case where all plaintext bytes are equal and all ciphertext bytes are equal.

**PlaintextVector**
Set only bit 0.

**Ciphertext0**
Set to a fixed value with all bytes equal

**Ciphertext1**
Same as Ciphertext0.

**SearchInfo**
UseCBC is set to 0. ExtraXOR is set to 1.

**PlaintextByteMask**
Set to 0x0F (only left half examined).

**PlaintextXorMask**
Set to 0x0000000000000000.

If the right and left half are equal, as must be the case if all plaintext bytes are the same, then when the ExtraXOR bit’s status causes the L=L XOR R step, L will become equal to 0. The plaintext byte mask selects only the left half and the PlaintextVector makes sure the 4 bytes are 0.

False positives occur whenever L=R, or with one key in $2^{32}$. Because this search is not guaranteed to terminate after $2^{56}$ operations, the average time is $2^{56}$ (not $2^{55}$). The number of false positives is expected to be $2^{56}/2^{32} = 2^{24} = 16.8$ million. Each search unit will thus find a false positive every $2^{32}$ keys on average, or about once every half hour. At 1 second polling of search units, $(0.5)(16.8 \text{ million})/3600 = 2333$ search unit hours will be idle (still under 1% of the total). The host computer will need to do the 16.8 million DES operations (on average), but even a fairly poor DES implementation can do this in just a few minutes.
**Scalability and Performance**

The architecture was intended to find DES keys in less than 10 days on average. The performance of the initial implementation is specified in Figure 2-4. Faster results can be easily obtained with increased hardware; doubling the amount of hardware will halve the time per result. Within the design, boards of keysearch ASICs can be added and removed easily, making it simple to make smaller or larger systems, where larger systems cost more but find results more quickly. Larger systems will have additional power and cooling requirements.

<table>
<thead>
<tr>
<th>Figure 2-4: Performance Estimate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total ASICs</td>
</tr>
<tr>
<td>Search units per ASIC</td>
</tr>
<tr>
<td>Total search units</td>
</tr>
<tr>
<td>Clock speed (Hz)</td>
</tr>
<tr>
<td>Clocks per key (typical)</td>
</tr>
<tr>
<td>DES keys per search unit per second</td>
</tr>
<tr>
<td>Total DES keys per second</td>
</tr>
<tr>
<td>Search size (worst case)</td>
</tr>
<tr>
<td>Seconds per result (worst case)</td>
</tr>
<tr>
<td>Days per result (worst case)</td>
</tr>
<tr>
<td>Search size (average case)</td>
</tr>
<tr>
<td>Seconds per result (average case)</td>
</tr>
<tr>
<td>Days per result (average case)</td>
</tr>
</tbody>
</table>

**Host Computer Software**

Cryptography Research will write the following software:

**Simulation**

Cryptography Research will develop software to generate test vectors for the chip for testing before the design is sent to the fab. This software will test all features on the chip and all modes of operation. This program will have a simple command line interface.

**Host computer**

The host computer software program will implement the standard search tasks of breaking a known plaintexts, breaking encrypted ASCII text (ECB and CBC modes), and solving the Matt Blaze challenge. These programs will be written in
standard ANSI C, except for platform-specific I/O code. The host program will also have a test mode, which loads search units with tasks that are known to halt reasonably quickly (e.g., after searching a few million keys) and verifies the results to detect of any failed parts. (The software will include the capability of bypassing bad search units during search operations.) Users who wish to perform unusual searches will need to add a custom function to determining whether candidate keys are actually correct and recompile the code.

The initial version of this program will have a simple command line interface and will be written for DOS. A Linux port will also be written, but may not be ready by the initial target completion date. (Because the only platform-specific code will be the I/O functions, it should be very easy to port to any platform with an appropriate compiler.) Software programs will identify the participants in the project (AWT, EFF, and Cryptography Research).

Cryptography Research will also produce a version with a prettier user interface to make the demonstration more elegant (platform-to-be-determined).

All software and source code will be placed in the public domain.

**Glossary**

**BoardID**
An 8-bit identifier unique for each board. This will be set with a DIP switch on the board. The host computer addresses chips by their ChipID and BoardID.

**CBC mode**
A DES mode in which the first plaintext block is XORed with an initialization vector (IV) prior to encryption, and each subsequent plaintext is XOR with the previous ciphertext.

**ChipID**
A value used by the host computer to specify which chip on a board is being addressed.

**Ciphertext**
Encrypted data.

**Ciphertext0**
The first of the two ciphertexts to be attacked.

**Ciphertext1**
The second of the two ciphertexts to be attacked.

Pre-ANSI C can be supported if required. Any GUI code will probably be written in C++. 
CiphertextSelector
A register used to select the current ciphertext being attacked. The selector is needed because a single DES engine needs to be able to test two ciphertexts to determine whether both are acceptable matches before deciding that a key is a good match.

DES
The Data Encryption Standard.

ExtraXOR
A register to make the search units perform an extra operation which XORs the right and left halves of the result together. This is used to add support for Matt Blaze’s DES challenge.

Host computer
The computer that controls the DES search array.

KeyCounter
Each search unit has a KeyCounter register which contains the current key being searched. These registers are each 7 bytes long, to hold a 56-bit key.

Plaintext
Unencrypted data corresponding to a ciphertext.

PlaintextByteMask
An 8-bit register used to mask off plaintext bytes. This is used to mask off bytes in the plaintext whose values aren’t known or are too variable to list in the PlaintextVector.

PlaintextVector
A 256-bit register used to specify which byte values can be present in valid plaintexts. It is the host computer’s responsibility to ensure that only a reasonable number of bits are set in the PlaintextVector; setting too many will cause the DES search units to halt too frequently.

PlaintextXorMask
A 64-bit register XORed onto the value derived by decrypting ciphertext 0. Normally this mask is either zero or set to the CBC mode initialization vector (IV).

SearchActive
A bit for each search unit which indicates whether it is currently searching, or whether it has stopped at a candidate key. Stopped search units can be restarted by loading a key which does not halt and resetting this bit.

SearchInfo
A register containing miscellaneous information about how DES results should be post-processed and also indicating whether any search units on the chip or on the
board have halted.

Use CBC
A bit in SearchInfo which directs the search engine to do CBC-mode post-processing after decryption (e.g., XOR the decryption of ciphertext1 with ciphertext0 to produce plaintext1).
In This chapter:
- ASIC Description
- Board description
- Read and Write Timing
- Addressing Registers
- All-active Signal
- ASIC Register Allocation

Design for DES
Key Search Array
Chip-Level Specification

Advanced Wireless Technologies, Inc.
and
Cryptography Research

ASIC Description

Select1
Selects Cipher text 1

C0
Cipher text 0

C1
Cipher text 1

Search
Search is active

K
Key

Mask
Plain text bit mask and DES output

Match=0
a Zero is found in any bit position of plain text vector as specified in step 4 of Search Unit Operation (see Chapter 2)

CBC & Extra XOR
Perform step 3 of Search Unit Operation (see Chapter 2)
To determine the maximum number of bit required for the Key:

\[ K = \log_2 (\text{Maximum combinations/number of chips}) \]

\[ = \log_2 \left( \frac{2^{56}}{(24 \text{ cpc} \times 64 \text{ cpb} \times 24 \text{ boards})} \right) = \log_2 (1.95E12) = 42 \text{ bits} \]

If we are going to use 32-bit counters, then it will overflow every:

\[ 2^{32} \times 16 \text{ cycles} \times 25\text{ns} = 1.72 \times 10^{12}\text{ns} = 1720 \text{ sec} = 28.7 \text{ minutes} \]
Board description

The PC will interface with the ASICs through a parallel card. The parallel card has three ports, assigned:

- Port A: Address(7:0)
- Port B: Data(7:0)
- Port C: Control, 8 signals

To reduce the routing resources on the boards and ASICs we multiplex the address lines. To access a register on the ASIC, it is required that the software latch the

---

Figure 3–2: State Diagram for the Search Unit
address three times: Board-ID(7:0), Chip-ID(6:0) and then Register address.

Having switches on the board makes the design flexible and expandable. Each board has its own unique Board-ID configured on switches: for example a board
with an ID of hexadecimal 5F has its board ID switches configured as follows:

![Board ID Switch Configuration](image)

**Read and Write Timing**

![Read and Write Timing Diagram](image)
**Chapter 3: Design for DES Key Search Array Chip-Level Specification**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{as1}$</td>
<td>10 ns</td>
<td>Min Board-ID and Chip-ID Address setup</td>
</tr>
<tr>
<td>$t_{as2}$</td>
<td>10 ns</td>
<td>Min Write Register-Address setup</td>
</tr>
<tr>
<td>$t_{as3}$</td>
<td>10 ns</td>
<td>Min Read Register-Address setup</td>
</tr>
<tr>
<td>$t_{ah1}$</td>
<td>10 ns</td>
<td>Min Board-ID and Chip-ID Address invalid (hold)</td>
</tr>
<tr>
<td>$t_{ah2}$</td>
<td>10 ns</td>
<td>Min Write strobe trailing edge to Address invalid (hold)</td>
</tr>
<tr>
<td>$t_{av}$</td>
<td>10 ns</td>
<td>Min ALE valid</td>
</tr>
<tr>
<td>$t_{ds}$</td>
<td>10 ns</td>
<td>Min Data valid to Write strobe goes low (setup)</td>
</tr>
<tr>
<td>$t_{ch}$</td>
<td>10 ns</td>
<td>Min Chip select hold</td>
</tr>
<tr>
<td>$t_{dh}$</td>
<td>10 ns</td>
<td>Min Write strobe goes high to data invalid (Data hold)</td>
</tr>
<tr>
<td>$t_{rv}$</td>
<td>10 ns</td>
<td>Min Read strobe duration</td>
</tr>
<tr>
<td>$t_{dv}$</td>
<td>100 ns</td>
<td>Max Read strobe goes low to data valid</td>
</tr>
<tr>
<td>$t_{dh}$</td>
<td>100 ns</td>
<td>Max Read strobe goes high to data invalid (Data hold)</td>
</tr>
</tbody>
</table>
Addressing Registers

Figure 3-4: Address Bus Scheme

All-active Signal

If low, one or more search unit is halted. This value is the result of ANDing all of the SearchActive bit together. We will place one AND gate per ASIC and cascade them.
## ASIC Register Allocation

<table>
<thead>
<tr>
<th>Address</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00-0x1f</td>
<td>PlaintextVector</td>
</tr>
<tr>
<td>0x20-0x27</td>
<td>PlaintextXorMask</td>
</tr>
<tr>
<td>0x28-0x2f</td>
<td>CipherText0</td>
</tr>
<tr>
<td>0x30-0x37</td>
<td>CipherText1</td>
</tr>
<tr>
<td>0x38</td>
<td>PlaintextByteMask</td>
</tr>
<tr>
<td>0x39-0x3e</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x3f</td>
<td>SearchInfo</td>
</tr>
</tbody>
</table>

### Additional Registers for Search Units

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40-0x47</td>
<td>Search Unit 0: Key counter (first 7 bytes) and Search Status</td>
</tr>
<tr>
<td>0x48-0x4f</td>
<td>Search Unit 1: Key counter (first 7 bytes) and Search Status</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xf8-0xff</td>
<td>Search Unit 23: Key counter (first 7 bytes) and Search Status</td>
</tr>
</tbody>
</table>

Number of register required:

58 common registers + 8 * n registers; n = the total number of search units in an ASIC

In this case n = 24, therefore 58 + 192 = 250 registers
Note: The unspecified pins are Non-Connects

CNTRL0 = ALE = ADDSEL1

CNTRL1 = CSB = ADDSEL2
The next few chapters of this book contain specially formatted versions of the documents that we wrote to design the DES Cracker. These documents are the primary sources of our research in brute-force cryptanalysis, which other researchers would need in order to duplicate or validate our research results.

**The Politics of Cryptographic Source Code**

Since we are interested in the rapid progress of the science of cryptography, as well as in educating the public about the benefits and dangers of cryptographic technology, we would have preferred to put all the information in this book on the World Wide Web. There it would be instantly accessible to anyone worldwide who has an interest in learning about cryptography.

Unfortunately the authors live and work in a country whose policies on cryptography have been shaped by decades of a secrecy mentality and covert control. Powerful agencies which depend on wiretapping to do their jobs — as well as to do things that aren't part of their jobs, but which keep them in power — have compromised both the Congress and several Executive Branch agencies. They convinced Congress to pass unconstitutional laws which limit the freedom of researchers — such as ourselves — to publish their work. (All too often, convincing Congress to violate the Constitution is like convincing a cat to follow a squeaking can opener, but that doesn't excuse the agencies for doing it.) They pressured agencies such as the Commerce Department, State Department, and Department of Justice to not only subvert their oaths of office by supporting these unconstitutional laws, but to act as front-men in their repressive censorship scheme, creating unconstitutional regulations and enforcing them against ordinary researchers and
The National Security Agency is the main agency involved, though they seem to have recruited the Federal Bureau of Investigation in the last several years. From the outside we can only speculate what pressures they brought to bear on these other parts of the government. The FBI has a long history of illicit wiretapping, followed by use of the information gained for blackmail, including blackmail of Congressmen and Presidents. FBI spokesmen say that was "the old bad FBI" and that all that stuff has been cleaned up after J. Edgar Hoover died and President Nixon was thrown out of office. But these agencies still do everything in their power to prevent ordinary citizens from being able to examine their activities, e.g. stonewalling those of us who try to use the Freedom of Information Act to find out exactly what they are doing.

Anyway, these agencies influenced laws and regulations which now make it illegal for U.S. crypto researchers to publish their results on the World Wide Web (or elsewhere in electronic form).

**The Paper Publishing Exception**

Several cryptographers have brought lawsuits against the US Government because their work has been censored by the laws restricting the export of cryptography. (The Electronic Frontier Foundation is sponsoring one of these suits, *Bernstein v. Department of Justice, et al*).* One result of bringing these practices under judicial scrutiny is that some of the most egregious past practices have been eliminated.

For example, between the 1970's and early 1990's, NSA actually *did* threaten people with prosecution if they published certain scientific papers, or put them into libraries. They also had a "voluntary" censorship scheme for people who were willing to sign up for it. Once they were sued, the Government realized that their chances of losing a court battle over the export controls would be much greater if they continued censoring books, technical papers, and such.

Judges understand books. They understand that when the government denies people the ability to write, distribute, or sell books, there is something very fishy going on. The government might be able to pull the wool over a few judges' eyes about jazzy modern technologies like the Internet, floppy disks, fax machines, telephones, and such. But they are unlikely to fool the judges about whether it's constitutional to jail or punish someone for putting ink onto paper in this free country.

* See http://www.eff.org/pub/Privacy/ITAR_export/Bernstein_case/.
Therefore, the last serious update of the cryptography export controls (in 1996) made it explicit that these regulations do not attempt to regulate the publication of information in books (or on paper in any format). They waffled by claiming that they “might” later decide to regulate books — presumably if they won all their court cases — but in the meantime, the First Amendment of the United States Constitution is still in effect for books, and we are free to publish any kind of cryptographic information in a book. Such as the one in your hand.

Therefore, cryptographic research, which has traditionally been published on paper, shows a trend to continue publishing on paper, while other forms of scientific research are rapidly moving online.

The Electronic Frontier Foundation has always published most of its information electronically. We produce a regular electronic newsletter, communicate with our members and the public largely by electronic mail and telephone, and have built a massive archive of electronically stored information about civil rights and responsibilities, which is published for instant Web or FTP access from anywhere in the world.

We would like to publish this book in the same form, but we can’t yet, until our court case succeeds in having this research censorship law overturned. Publishing a paper book’s exact same information electronically is seriously illegal in the United States, if it contains cryptographic software. Even communicating it privately to a friend or colleague, who happens to not live in the United States, is considered by the government to be illegal in electronic form.

The US Department of Commerce has officially stated that publishing a World Wide Web page containing links to foreign locations which contain cryptographic software “is not an export that is subject to the Export Administration Regulations (EAR).” This makes sense to us — a quick reductio ad absurdum shows that to make a ban on links effective, they would also have to ban the mere mention of foreign Universal Resource Locators. URLs are simple strings of characters, like http://www.eff.org; it’s unlikely that any American court would uphold a ban on the mere naming of a location where some piece of information can be found.

Therefore, the Electronic Frontier Foundation is free to publish links to where electronic copies of this book might exist in free countries. If we ever find out about such an overseas electronic version, we will publish such a link to it from the page at http://www.eff.org/pub/Privacy/Crypto_misc/DES_Cracking/.

* In the letter at http://samsara.law.cwru.edu/comp_law/jvd/pdj-bxa-gjs070397.htm, which is part of Professor Peter Junger’s First Amendment lawsuit over the crypto export control regulations.
Scanning

When printing this book, we used tools from Pretty Good Privacy, Inc (which has since been merged into Network Associates, Inc.). They built a pretty good set of tools for scanning source code, and for printing source code for scanning. The easiest way to handle the documents we are publishing in this book is to use their tools and scanning instructions.


The tools and instructions from the OCR Tools book are now available on the Internet as well as in PGP’s book. See http://www.pgpi.com/project/, and follow the link to “proof-reading utilities”. If that doesn’t work because the pages have been moved or rearranged, try working your way down from the International PGP page, http://www.pgpi.com.

PGP’s tools produce per-line and per-page checksums, and make normally invisible characters like tabs and multiple spaces explicit. Once you obtain these tools, we strongly suggest reading the textual material in the book, or the equivalent README file in the online tool distribution. It contains very detailed instructions for scanning and proofreading listings like those in this book. The instructions that follow in this chapter are a very abbreviated version.

The first two parts of converting these listings to electronic form is to scan in images of the pages, then convert the images into an approximation of the text on the pages. The first part is done by a mechanical scanner; the second is done by an Optical Character Recognition (OCR) program. You can sometimes rent time at a local “copy shop” on a computer that has both a scanner and an OCR program.

When scanning the sources, we suggest “training” your OCR program by scanning the test-file pages that follow, and some of the listings, and correcting the OCR program’s idea of what the text actually said. The details of how to do this will depend on your particular OCR program. But if you straighten it out first about the shapes of the particular characters and symbols that we’re using, the process of correcting the errors in the rest of the pages will be much easier.

Some unique characters are used in the listings; train the OCR program to convert them as follows:
Right pointing triangle (used for tabs) - currency symbol (byte value octal 244)

Tiny centered triangle “dot” (used for multiple spaces) - center dot or bullet (byte value octal 267)

Form feed - yen (byte value octal 245)

Big black square (used for line continuation) - pilcrow or paragraph symbol (byte value octal 266).

Once you’ve scanned and OCR’d the pages, you can run them through PGP’s tools to detect and correct errors, and to produce clean online copies.

**Bootstrapping**

By the courtesy of Philip R. Zimmermann and Network Associates, to help people who don’t have the PGP OCR tools, we have included PGP’s bootstrap and bootstrap2 pages. (The word *bootstrap* refers to the concept of “pulling yourself up by your bootstraps”, i.e. getting something started without any outside help.) If you can scan and OCR the pages in some sort of reasonable way, you can then extract the corrected files using just this book and a Perl interpreter. It takes more manual work than if you used the full set of PGP tools.

The first bootstrap program is one page of fairly easy to read Perl code. Scan in this page, as carefully as you can: you’ll have to correct it by hand. Make a copy of the file that results from the OCR, and manually delete the checksums, so that it will run as a Perl script. Then run this Perl script with the OCR result (with checksums) as the argument. If you’ve corrected it properly, it will run and produce a clean copy of itself, in a file called *bootstrap*. (Make sure none of your files have that name.) If you haven’t corrected it properly, the perl script will die somehow and you’ll have to compare it to the printed text to see what you missed.

When the bootstrap script runs, it checks the checksum on each line of its input file. For any line that is incorrect, the script drops you into a text editor (set by the EDITOR environment variable) so you can fix that line. When you exit the editor, it starts over again.

Once the bootstrap script has produced a clean version of itself, you can run it against the scanned and OCR’d copy of the bootstrap2 page. Correct it the same way, line by line until bootstrap doesn’t complain. This should leave you with a clean copy of bootstrap2.

The bootstrap2 script is what you’ll use to scan in the rest of the book. It works like the bootstrap script, but it can detect more errors by using the page
checksum. Again, it won't correct most errors itself, but will drop you into an editor to correct them manually. (If you want automatic error correction, you have to get the PGP book.)

All the scannable listings in this book are in the public domain, except the test-file, bootstrap, and bootstrap2 pages, which are copyrighted, but which Network Associates permits you to freely copy. So none of the authors have put restrictions on your right to copy their listings for friends, reprint them, scan them in, publish them, use them in products, etc. However, if you live in an unfree country, there may be restrictions on what you can do with the listings or information once you have them. Check with your local thought police.
This is a test page for OCR training. This includes many possible
characters:

1. Tabs
2. Spaces
3. Some normally non-printing characters are printed.
4. A very long line is wrapped as follows:
as f3d7a !"%&'()*+,-./0123456789:;<=?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~

Very long lines are wrapped as follows:
a5f3d7a !"%&'()*+,-./0123456789:;<=?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
2abc00 abcdefghijklmnopqrstuvwxyz|abcdefghijklmnopqrstuvwxyz|
2abc02 abcdefghijklmnopqrstuvwxyz|abcdefghijklmnopqrstuvwxyz|
a12abc abcdefghijklmnopqrstuvwxyz|abcdefghijklmnopqrstuvwxyz|

The following lines have 77 underscores:

Chapter 4: Scanning the Source Code
--e140 001c21fe2a2e08010002 Page 1 of bootstrap
94e666 #!/usr/bin/perl -s
0e69b1 #
754667 # bootstrap -- Simpler version of umunge for bootstrapping
0e8a601 #
9f3339 # Unmunge this file using:
313c3 # perl -ne if ($/ ^ E"[-\s]$(4,6) /( /s/[244][245][267]/g; print; )
29a601 #
438bb8 # $Id: bootstrap,v 1.15 1997/11/14 03:52:53 mwh Exp $
85a5f5a
85a49b sub Fatal { print STDERR "\$\_ = exit(1); "
50a97 sub Max { my ($a, $b) = \$\_; \$a > \$\_ ? \$a : \$b }
3e6b1 sub TabSkip { $tabWidth - 1 - (Length($\_[0]) \% $tabWidth) }
af5a5a
3a5oe ($stab, $yen, $silc, $cdot, $tmp1, $tmp2) = ('"244", "245", "266", "267", "377", "376");
393067 $editor = %ENV{'VISUAL'} || %ENV{'EDITOR'} || 'vi';
d3e7e6 $infile = $ARGV[0];
d9e42f doFile: {
435e9b ....open(IN, "<$inFile") || die;
095163 .... for ($lineNum = 1; ($\_ = <IN>) < 0; $lineNum++) {
7daaad s/'\s+//; s/'\s+$/;' 
5b52f next if ('*/'); # Ignore blank lines
53a11 ($prefix, $seenCRCStr, $dummy, $\_) = /^\$2\$(4)(\.*)?$/
6a5f5a
b2ba8c # Correct the number of spaces after each tab
a32d31 while (s/$tab(*)/$tmp1 . ($tmp2 x $Max(length($\_), $TabSkip($' )))/e) {
5c9e47 ($\_2) = s/$/; s/$/; s/$/; # Correct center dots
a0024c c = $tmp1/$tab/g; s/$/; s/$/; # Restore tabs and spaces from correction
83b3cd s/$\_*/
2ba5f # Strip trailing spaces, and add a newline
9f215d $crc = $seenCRC = 0;
3f3db3 for ($data = $\_; $data ne "") {
68e002 .... $crc ^= ord($data);
342e8 .... for (1..8) ({
0e8b86 .... $crc = ($crc >> 1) ^ ($crc & 1) ? 0x8040 : 0;
2ca5da ....)
5a1aea )
264656 if ($crc != hex($seenCRCStr)) {
42b97 .... close(IN); close(OUT);
a94221 .... unlink(@filesCreated);
e4cf5 .... @filesCreated = ();
4fcee2 .... @oldStat = stat($inFile);
b9e5b0 .... system($editor, "$lineNum", "$inFile");
d34e59 .... @newStat = stat($inFile);
b9d9c5d .... redo doFile if ($oldStat[0] != $newStat[0];
68a0de .... &Fatal("Line $lineNum invalid: $\_");
7e1aea }
3d45a
f3a5f5a a97b7d if ($prefix eq '---') { \# Process header line
3d454c .... ($code, $pageNum, $fFile) = /^\$(19)$/ Page (\d+) of (.*);
2e88c3 .... $tabWidth = hex(substr($code, 11, 1));
c77d01 .... if ($fFile ne $lastFile) {
9d1aaf print "$fFile";\n;
9e8be7 .... &Fatal("$fFile: already exists\n") if (!$f \& (-e $fFile));
eaab63 .... close(OUT);
db987b .... open(OUT, "$\_2") || &Fatal("$fFile: $!\n");
4cb2d1 .... push(@filesCreated, ($lastFile = $fFile));
3c45d ....)
3d44cd } else { \# Unmunge normal line
2e2591 .... $tab(*) = "\t"." x (Length($\_1) - $tabWidth))/eg;
47b547 .... s/$ynvu/\n/; \# Handle form feeds
5c17d4 .... s/$silc/\n/; \# Handle continuation lines
019f62 .... s/$cdot/ /g; \# Center dots -> spaces
23a5f5a 97e546 .... print OUT;
591eaa }
4f6f67 ....)
1cc0a6 .... close(IN); close(OUT); c2e6e6
This chapter contains a complete listing of the C-language software that we wrote to control the DES Cracker hardware. This software provides a simple user interface for testing the hardware, setting up problems to be solved by searching through the possible keys, and running such searches. We're publishing it to show both people and machines how to control the DES Cracker.

This version of the software is fairly rudimentary; it doesn't include a graphical user interface, collaborate with others across the Internet to speed up brute force cracking attempts, etc. By the time you read this book, there will probably be a better version of the software, which you will be able to read about in our web pages at http://www.eff.org/pub/Privacy/Crypto_misc/DES_Cracking/.

This software is known to build and run in a “DOS Window” under Windows 95 on a PC using the Borland C++ Compiler, version 3.1. It also compiles cleanly using Microsoft Visual C++ version 5.

The software is documented in the file readme.txt.

For details on why these documents are printed this way, and how to scan them into a computer, see Chapter 4, Scanning the Source Code.
<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>SHA-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANIFEST</td>
<td>1</td>
<td>7bf681</td>
</tr>
<tr>
<td>readme.txt</td>
<td>2</td>
<td>ec8ce7</td>
</tr>
<tr>
<td>autoconf.c</td>
<td>3</td>
<td>05e777</td>
</tr>
<tr>
<td>build.bat</td>
<td>4</td>
<td>ced6b1</td>
</tr>
<tr>
<td>chipio.c</td>
<td>5</td>
<td>a0056e</td>
</tr>
<tr>
<td>chipio.h</td>
<td>6</td>
<td>151f75</td>
</tr>
<tr>
<td>des.c</td>
<td>7</td>
<td>1a8e30</td>
</tr>
<tr>
<td>des.h</td>
<td>8</td>
<td>11db2a</td>
</tr>
<tr>
<td>initsrch.c</td>
<td>9</td>
<td>0aa3f</td>
</tr>
<tr>
<td>keyblock.c</td>
<td>10</td>
<td>0f78d8</td>
</tr>
<tr>
<td>keyblock.h</td>
<td>11</td>
<td>f699dd</td>
</tr>
<tr>
<td>search.c</td>
<td>12</td>
<td>ba96cf</td>
</tr>
<tr>
<td>search.h</td>
<td>13</td>
<td>5127f6</td>
</tr>
</tbody>
</table>
--e562 000445155728000002 Page 1 of readme.txt

The configuration program is an important part of the DES Cracker. Because there are a large number of chips in the system, it is inevitable that a few fail. By automatically removing defective units, it is not necessary to repair the system when failures do occur.

The program "autoconf.exe" will automatically identify the configuration of a search array. With the I/O port base address at 210 hex, simply run the program with the command:

```
  cd7d24 -t
```

Note that the "-t" flag performs register testing (recommended if the search system might contain defective chips that need to be avoided). The "-v" flag provides verbose output. For example:

```
  cd7d24 -t -v
```

When autoconf completes, it will print the total number of chips to the screen and save the configuration information to the configuration file. The configuration can be edited (e.g., with grep) to remove defective units not caught with autoconf.

(Not that this first release does not implement search unit testing; code except for the register tests.)

The search parameters have to be specified before a key can be found. The program initsrch creates a "search context" file that contains these search parameters and a list of the regions of keyspace that occur.

grep)
Chapter 5: Software Source Code

--8d6f 0009a4c5f7080020002 Page 2 of readme.txt

6d2d23 remain to be searched.
f3af5a 757db9 The search parameters can either be entered into initsrch or
03be37 specified on the command line. To enter them manually, run initsrch
b172e9 with no parameters:
34af5a f0a5ad --> initsrch
da4af5a 83b591 The program will then prompt for the search context file. Press
a473ca for the default filename ("search.ctx").
98af5a 953b91 Next, the program will prompt for a search mode. Five modes are:
0609f8 supported and are described in the following sections.
30af5a 45b5a1 ... K - Known plaintext
42856 ... E - ECB ASCII text
019f99 ... C - CBC ASCII text
31e1fd ... B - Blaze challenge
ddb6b9 ... M - Manual parameter specification
e9af5a 5af5a 2edf4c 1. Known plaintext searching
ccaf5a 06f1ec This is the simplest (and most common) mode of operation. If a
4dd2e9 complete DES plaintext/ciphertext pair is known, this mode can be
073f1b used to quickly search for the key. When prompted, enter the
c281eb plaintext in hexadecimal form (e.g., "123456789ABCD0F") and press
95792f enter. Next, enter the ciphertext, also in hexadecimal. The program
6fb0ea will then create a search context file and exit.
daf5a fc5a 2. ECB ASCII text searching
9aaf5a 66c8f1 If your target message is known to be ASCII text and was encrypted
1edf6 using DES ECB mode, enter two different ciphertexts. The program
d92df0 will create the search context file and exit. The program is
234bf0 configured to include all letters ("a-z" and "A-Z"), numbers ("0-9"),
478b54 and common punctuation (ASCII zero, tab, linefeed carriage return,
0ded96 space, and common punctuation ("," , , , , , ) . For other character
98f3b6 sets, use the manual parameter specification option.
1e2a5a bc5a 3. CBC ASCII text searching
caf5a a7fe29 If your message is ASCII text and was encrypted using DES CBC mode,
fb5a5e this option lets you specify an initialization vector and two
a45deo ciphertext messages. The CBC mode ASCII option uses the same ASCII
6c4548 text characters as ECB ASCII.
09af5a 18af5a 70a6d7 4. The Blaze challenge
0caf5a 20b2e8 Matt Blaze's DES challenge involves searching for a key such that a
1b8f5e repeated plaintext byte produces a repeated ciphertext byte. This
2851f0 option will search for keys that meet the challenge. Simply specify
the desired repeated ciphertext byte.
6ba5a 6ba5a fb0aeb 5. Manual parameter specification
e2af5a f4f30d The manual parameter mode allows direct control over the search
a1c83a parameters. The manual mode requires entering more data than the
w1978 other modes; it is often easier to pipe input from a script file,
3acabf e.g.:
6caf5a 11ce2d --> initsrch < search.scr
5ba5a 2b6c92 First, enter the plaintext vector. This is 64 hex digits long and
984f44 specifies the bytes that can appear in "valid" plaintexts. The most
09af5a significant bit of the left-hand digit specifies whether ASCII 255
ea0906 can appear, and the least significant bit of the last digits specifies
9ad545 whether ASCII zero can appear. For example, the plaintext vector for
569be the ASCII text modes is:
45af5a
328242 ...00000000000000000000000007FFFFFFC7FFFFFFE8FFF73870002601
47af5a
629a4f Next, enter the initialization vector for the first DES, if any.
68bf16 This will be XORed onto the first plaintext before its validity is
5587b3 checked.
6baf5a
f4fd67 Next, enter the two ciphertexts (ciphertext 0 and ciphertext 1).
aaf9a7 These may be the same or different.
a2af5a
d4a2ed Next, enter the plaintext byte mask. This sets bits that should be
f36e4e ignored in the plaintext. For example, if the left-hand byte of the
ca85d plaintext is unknown or can have any value, the plaintext byte mask
379f187 would be set to 80 (hex).
11af5a
9d87cc finally, enter the searchInfo byte. Bit 1 of this byte specifies
97af25 whether CBC mode should be used. If so, the first ciphertext will be
595e4f XORed onto candidate plaintexts produced by decrypting the second
2d5f09 ciphertext. Bit 2 of searchInfo specifies whether the extraXor
13de86 operation should be done. This operation XORs the right half of the
8aacc2 plaintext onto the left half before it is checked. (For the Blaze
06de4f challenge, the desired plaintext has a single byte repeated. The
13ab4a extraXor operation will set the left half of the plaintext to zero if
c6781e the plaintext is good. The plaintextByteMask can then be set to 0x0F
5217de to ignore the right half and the plaintextVector has only the bit for
3af5a
1baf5a
4c28ed 5. The search context file
8daf5a
179dcb The search context file contains a header, the search parameters, and
8f1477 2^24 bits corresponding to the unsearched key regions. The search
91edf parameters are: plaintextVector (32 bytes), plaintextXorMask (8
5dd722 bytes), ciphertext0 (8 bytes), ciphertext1 (8 bytes),
3583fb plaintextByteMask (1 byte), and searchInfo (1 byte). Each search
421808 region includes 2^32 keys. The first bit (the MSB of the first key
b79fb8 region byte) corresponds to the keys 000000000000000 through
4e2847 000000FFFFFFF, in 56-bit notation. (To produce the 56-bit form of a
f21751 64-bit DES key, delete the eight parity bits.)
d6af5a
efaf5a
f5af5a
48176f ----------------------------- 80000000000000000000000000000000
81999d Section 4: Running a Search.
c1af5a
05ae2f The most common way to run a search is to type:
5caf5a
057f42 ...> search search.cfg search.ctx logfile -q
27af5a
80ed9 The "-q" flag requests quiet output, which prints less information to
e10f1f the screen. The search.cfg file is produced by autoconf, and
2200bc search.ctx is produced by ointsrch. The logfile will contain a list
f4a0ec of candidate keys encountered.
12af5a
223e7f If a search is stopped partway through, work done in partially-
0cb69 completed key regions is lost, but completed regions are noted in the
a252e search context file. Note that a complete search will produce a
433e44 rather large amount of data in the logfile. If hard disk space is
649466 limited, it may be desirable to stop the search occasionally (for
9ad667 example, daily) to purge the logfile.
c2af5a
5af5a
1eaf5a
8b176f ----------------------------- 80000000000000000000000000000000
8e845d Section 5: Porting to other platforms.
d9af5a
9aa861 When porting to other platforms, some code changes or additions may
9a17dd be required. The following may not be found on all systems:
d9af5a
094ef6 ...stricmp ... This is a case-insensitive strcmp found on many
659851 ... compilers. If it isn't present, you can either use strcmp
Chapter 5: Software Source Code

--a334 0019f8825ad80020002 Page 4 of readme.txt

70faf5 .......... (though commands will become case sensitive) or write onc.
4daf5a
57970a .......... SEEK_SET: A constant (equal to zero) used to tell fseek()
bdc7f8 .......... to go to a fixed offset. Usually defined in stdio.h
02af5a
ca3e28 .......... kbhit(void): Returns true if a key has been pressed. (Used to
6621f1 .......... check for commands during searches.)
bbaf5a
6d2832 .......... getch(void): Reads a keystroke from the keyboard.
4daf5a
13802 .......... inportb(unsigned portNum): Reads a byte from an I/O port. Used
5c27f1 .......... only by chipio.c. On other platforms, inportb may need to
27977e .......... be emulated. (For Visual C++, inportb is implemented in
3c461d .......... chipio.c as inline assembly language.)
97af5a
9e58f5 .......... outportb(int portNum, int value): Sends a byte to an I/O port.
59dbb1 .......... Used only by chipio.c. On other platforms, outportb may
3bb05f .......... need to be emulated. (For Visual C++, outportb is
0f88ab .......... implemented in chipio.c as inline assembly language.)

As this code goes to press, there was little opportunity for testing
and the code has not undergone any of the assurance, code review, or
testing processes we normally use. When working on the code, you
you may find a few bugs. Feedback, as always, is appreciated.

Paul Kocher, Josh Jaffe, and everyone else at Cryptography Research
would like to thank John Gilmore and the EFF for funding this unique
project, and AWT for their expert hardware work!
---8884 0008b9a267780020003 Page 1 of autoconf.c

8d2d83 #define SOFTWARE_VERSION "1.0"
9a625c #define SOFTWARE_DATE "04-21-1998"
86a65a

d7a75a #define MAX_CHIPS_PER_BOARD 64
f191af #define MAX_BOARDS 256
59a45a

e9708e static void EXIT_ERR(char *s) { fprintf(stderr, %s); exit(1); }
619181 void AutoconfigureScan(FILE *fp, int fullScan, int verbose);
ca05e4 int QuickCheckRegister(int board, int chip, int register, int value);
cc0c65 void AddSearchUnits(FILE *fp, int board, int chip, int unit, int isGood);
69635c long DoFullScan(FILE *fp, int board, int* chips, int verbose);
7a7a5a

fc0a5a int main(int argc, char **argv) {
1c3580

d3a587 printf("\nUsage: autoconf search.cfg [baseIoPort] [-v] [-t#]\n"
4245d9 printf("\n---\n"
8f0f87 printf("\n---\n"
9c0c8e printf("\n---\n"
16a08a printf("\n---\n"
4b0dd7 printf("\n---\n"
8b1854 printf("\n---\n"
--bbbf 000364123980020003 Page 2 of autoconf.c
d1805d ....

Chapter 5: Software Source Code
---

5aa1fb .... SOFTWARE_VERSION, SOFTWARE_DATE);
5daaf5a
aebc69a if (argc < 2) argv[1][0] = '-'
a7a829b EXIT-ERR(helmpMsg);
0a3e7fe fileSpec = argv[1];
bab35d for (nextArg = 2; nextArg < argc; nextArg++) {
77f71b if (argv[nextArg][0] == '-') {
83c9ff if (toupper(argv[nextArg][1]) == 'Y') {
11f2647 sscanf(argv[nextArg]+2, "%d", &testLoops);
1fb4e1 if (testLoops < 0) dca4df testLoops = 0;
4cbe96 else if (toupper(argv[nextArg][1]) == 'V') 1e568e verbose = 1;
78a69a else if (argc < argvC negate = EXIT-ERR("Bad parameter (run with no parameters for help)\n");
6a6a79 else {
869b6 sscanf(argv[nextArg], "%x", &baseIoPort);
7c72e3 if (baseIoPort == 0) 1e8d34 EXIT-ERR("Bad parameter (run with no parameters for help)\n"); b76e7e ...
31f1c ...
41a5f
a3a98a if (verbose) printf("Test parameters:\n");
190eb3 if (verbose) printf(" BaseIoPort = %x \n", baseIoPort);
7ae35e if (verbose) printf(" outputfile = \"%s\" \n", fileSpec);
d9d929 if (verbose) if (testLoops < 0) printf(" Quick scan only\n"); b60935 if (verbose) if (testLoops == 0) printf(" Full register scan\n"); b9142f if (verbose) if (testLoops > 0) printf(" %d DES tests\n", testLoops);
d9af5a
6407e8 fp = fopen(fileSpec, "w");
d9d929 if (fp == NULL) a8ee0b EXIT-ERR("Error opening output file \n");
7a575c fprintf(fp, "%s Auto-generated search system config file\n"); 98932f fprintf(fp, "PORT=%x \n", baseIoPort);
baa5f
253c81 SetBaseAddress(baseIoPort);
d9aaf7 fullScan = (testLoops < 0) ? 0 : 1; 
91bad4 AutoconfigureScan(fp, fullScan, verbose);
aced24 fclose(fp);
faaf5a 368f0d for (i = 0; i < testLoops; i++) {
5b2c130 printf("Doing DES test %d of \%d \n", i+1, testLoops);
62bb1a fp = fopen(fileSpec, "w");
a541c7 if (fp == NULL) 
0a6495 EXIT-ERR("Error reopening output file \n");
864529 fgets(buffer, 190, fp); /* skip header line */ 
e19a5b fgets(buffer, 190, fp); /* skip port line */ 5d2f5d fprintf(stderr, " *** Detailed test not implemented !!!\n"); b0b17e fclose(fp);
65df1c ...
91c86a return (0);
57ef6e ...
bdaf5a e4a5f ...
8da5f ...
e2385e2 /* Automatically figure out the configuration of the search system.
6d6bac2 */ Thus function assumes that SetBaseAddress() has already been called.
6d4952 /*
841b6d void AutoconfigureScan(FILE *fp, int fullScan, int verbose) {
6e4b07 int board, chip, chipCount, value;
5b7b20 long totalChips = 0;
65802b int chips[MAX_CHIPS_PER_BOARD];
2ba5fa ...
dca3b4 if (verbose) printf(" *** DOING AUTOCONFIGURE SCAN ***\n"); 
f1937a8 for (board = 0; board < MAX_BOARDS; board++) {
88df2e printf("CHECKING BOARD 0x%02x: \", board);
6d442f fflush(stdout);
3680db chipCount = 0;
34211c for (chip = 0; chip < MAX_CHIPS_PER_BOARD; chip++) {
/* TEST FIRST BYTE OF CIPHERTEXT 0 (REGISTER 0x28) */
79c8a7 if (QuickCheckRegister(board, chip, 0x28, value) == 0) {
5acc0c QuickCheckRegister(board, chip, 0x28, value=255) == 0) {
78875d chips[chip] = 0;
8c328b if (verbose) printf("\n \ \BOARD 0x%02X CHIP 0x%02X: Not found.",
e9d06e board, chip);
1b9f97 else {
41dd81 chips[chip] = 1;
dad3c2 chipCount++;
f287c8 if (verbose) printf("\n \ \BOARD 0x%02X CHIP 0x%02X: FOUND", board, chip);
903982 if (fullScan) {
b2986b if (verbose) printf("\n \ \CHIP 0x%02X: Halting chip for test", chip);
61fd4e SetRegister(board, chip, REG_PTXT_BYTE_MASK, 0xFF);
d17fb0 }
8842cc }
e96fe7 }
47f418 if (verbose) printf("\n");
347082 printf("Found %d chips total.\n", chipCount);
90a4fa /* DO DETAILED REGISTER SCAN IF REQUESTED */
e9eef1 if (fullScan && chipCount) {
45f8b8 totalChips = DoFullScan(fp, board, chips, verbose);
e46a79 else {
0d79b chipCount = 0;
e2f9e5 for (chip = 0; chip < MAX_CHIPS_PER_BOARD; chip++) {
77b833 if (chips[chip]) {
b0d98b chipCount++;
2b3738 totalChips++;
065ead AddSearchUnits(fp, board, chip, -1, 1);
237fb0 }
0d42cc }
316ef7 }
f5d1f1c 21397d if (verbose) printf("*** AUTOCONFIGURE SCAN COMPLETE ***\n");
05c77d printf("Found %ld chips total.\n", totalChips);
daef63 }
1f1a5a 66ddaa int QuickCheckRegister(int board, int chip, int reg, int value) {
1c3f5a 
66ddaa int QuickCheckRegister(int board, int chip, int reg, int value) {
1f1a5a 
66ddaa int QuickCheckRegister(int board, int chip, int reg, int value) {
1c3f5a 
66dd 80c945 void AddSearchUnits(FILE *fp, int board, int chip, int unit, int isGood) {
571f0e } return (0); 9bf30e if (unit < 0) {
e154b0 for (i = 0; i < SEARCH_UNITS_PER_CHIP; i++)
265e78 AddSearchUnits(fp, board, chip, i, 1);
3349d8 else {
1f1ac0 printf(fp, "%s=0x%02X 0x%02X 0x%02X\n", isGood ? "UNIT" : "FAIL",
1129d9 board, chip, unit);
5ad1c }
9befe6 }
faef5a 
0aef5a 
777b47 long DoFullScan(FILE *fp, int board, int* chips, int verbose) {
8eff49 int chip, reg, seed, value, i, j;
33160d int units[24];
0fb920 long totalChips = 0;
caef5a 24d7a0 if (verbose) printf(" \ --- Register scan on board 0x%02X --\n", board);
deaf5a 61377a /* PICK A SEED & USE IT TWICE (ONCE WHEN SETTING & ONCE WHEN CHECKING */
a172a5 seed = (int)time(NULL);
64af5a 
bb775e /* SET registers */
8ad8d srand(seed);

Chapter 5: Software Source Code

```c
--ff7e 000105deab4880020003 Page 4 of autoconf.c

--ff7e 000105deab4880020003 Page 4 of autoconf.c

bc3f58 ...for (chip = 0; chip < MAX_CHIPS_PER_BOARD; chip++) {

d05448 ...if (chips[chip] == 0)
325f6a ...continue;

e8c9858 ...if (verbose) printf(" ...BOARD 0x%02X CHIP 0x%02X: Setting reg...

d05b4e ...-board, chip);

ecab177 ...for (reg = 0; reg <= 0xFF; reg++) {

e5beec ...if (((reg >= 0x39 && reg < 0x40) || (reg > 0x40 && (reg & 7) == 7))
7ca511 ...continue;

ecce87b ...value = rand() & 255;
9aca50 ...SetRegister(board, chip, reg, value);

7e6fe7 ...
```

5-10
--4dcb 001b8acf45a800020004 Page 1 of build.bat

eb1685 rem Sample build script (using Microsoft Visual C++)
89af5a
a05793 cl search.c keyblock.c chipio.c des.c
049d12 cl initsrc.c keyblock.c
818947 cl autoconf.c chipio.c
110940 cl testvec.c sim.c des.c
b8af5a
---27e 000793f2b7c80020005 Page 1 of chipio.c

8d2d03 /*******************************************************
a07c89 * chipio.c*************************************/
87350a * Search Engine Low-Level Hardware Interface Module
be29eb * **************Paul Kocher for the Electronic Frontier Foundation (EFF).*
b2aaf * Placed in the public domain by Cryptography Research and EFF.*
aecaeb * THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK.*
6c4992 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM.*
6d29eb * Revision History:
0e29eb * Version 1.0: Initial release by Cryptography Research to EFF.*
3dd85a * Written 1998 by Cryptography Research (http://www.cryptography.com)*
596ef * 1998, Paul Kocher.
2f128d9 * HISTORY:
8d29eb * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM.*
3dd85a * Version 1.0: Initial release by Cryptography Research to EFF.*
596ef * 1998, Paul Kocher.

49a5fa 4c8eb2 #include <stdio.h>
24a5fa #include <conio.h>
4e1519 #include "chipio.h"

24a5fa static int CURRENT_BOARD = -1;
3c611c static int CURRENT_CHIP = -1;
0c1693 static int CURRENT_PORT_CONF = -1;
766981 static int IO_BASE_ADDRESS = 0x210;

29a5fa #define IO_PORTA_ADDRESS (IO_BASE_ADDRESS+0)
823310 #define IO_PORTB_ADDRESS (IO_BASE_ADDRESS+1)
401c87 #define IO_PORTC_ADDRESS (IO_BASE_ADDRESS+2)
5f8cee #define IO_CNFG_ADDRESS (IO_BASE_ADDRESS+3)
7d972f #define CNFG_OUTPUT 0x80
b52497 #define CNFG_INPUT 0x82

5a5fa 3c951 #define CTRL_BASE 0x10 /* base value onto which others are XORed */
663867 #define CTRL_RST 0x20

e7dce1 #define CTRL_RDB 0x10
8a6735 #define CTRL_WRB 0x88
777b8e #define CTRL_ALE 0x04

dd9757 #define CTRL_ADRSEL2 0x02 /* in documentation is also called CTN1 */
e884f1 #define CTRL_ADRSEL1 0x01 /* in documentation is also called CTN0 */

49a5fa 4ce988 /* DELAYS CAN BE ADDED TO DEAL WITH BUS LOADING/CAPACITANCE/ETC.
59495d */

aee62f #define DELAY_FACTOR 100L
67ac33 #define DELAY_ADDRESS_SETTLE *DELAY_FACTOR
ecf61f #define DELAY_DATA_SETTLE *DELAY_FACTOR
b21012 #define DELAY_RST_HOLD *DELAY_FACTOR
3a8807 #define DELAY_RST_RECOVER *DELAY_FACTOR
c48418 #define DELAY_RDB_HOLD *DELAY_FACTOR
5d2f12 #define DELAY_RDB_RECOVER *DELAY_FACTOR
439b05 #define DELAY_WRB_HOLD *DELAY_FACTOR
753210 #define DELAY_WRB_RECOVER *DELAY_FACTOR
97157a #define DELAY_ALE_SETTLE *DELAY_FACTOR
8708a #define DELAY_ADRSEL2_SETTLE *DELAY_FACTOR
b987e #define DELAY_ADRSEL1_SETTLE *DELAY_FACTOR

eca5fa #define iodelay(delayTime) () { } /* insert delay if rqd */

d5a5fa 08a45a

fcc96f #ifdef _MSC_VER
f34ce5 /* Microsoft C++ Direct I/O functions
b0495d */

aa42df static int inportb(int portNum) {
459b00 /* unsigned char rval;
5a8ce0 /* unsigned short portNumShort = (unsigned short)portNum;
ac5fa
7948ba /*asm { mov dx, portNumShort }

---
```
--8oc3 00063846a280020005 Page 2 of chipio.c
05a1b8 ...asm ( in dx, al )
390458 ...asm ( mov rva, al )
5b76c9 ...return ( rva );
```

```c
59af5a
019285 static void outportb(int portNum, int val) {
3355b6 unsigned char valChar = (unsigned char)val;
08b86d unsigned short portNumShort = (unsigned short)portNum;
64af5a
ac005a ...asm ( mov dx, portNumShort )
6893dc ...asm ( mov al, valChar )
1affc5 ...asm ( out dx, al )
a5efe6 }
987454 #endif
e8af5a
f0af5a
04d629 static void ConfigureI0Port(int inputOrOutput) {
37df70 ...outportb(IO_CNFG_ADDRESS, inputOrOutput);
a33113 CURRENT_PORT_CNFG = inputOrOutput;
7eaf5a
226c42 ...* Warning:
022fb8 ...*
38d9be ...* Changing the I0 port state causes a tiny glitch to go out on the
61e818 ...* PC-DIO card. This is enough to occasionally trigger the ALE, which
073299 ...* causes read/write errors. To avoid this, always explicitly
dcbbe3 ...* re-select the chip after switching port directions.
8c9af5a ...*/
96e4d4 CURRENT_CHIP = -1;
71efe6 }
dcaf5a
bba550 static void SetAddress(int addressValue) {
334e16 outportb(IO_PORTA_ADDRESS, addressValue);
85ef6 }
1ca5a
49af5a
2952e2 static void SetData(int dataValue) {
e81c12 ...outportb(IO_PORTB_ADDRESS, dataValue);
20ef6 }
6af5a
08af5a
1db8ab static int GetData(void) {
5899a6 ...return (inportb(IO_PORTB_ADDRESS));
67efe6 }
98af5a
eba550 static void SetControl(int controlPortValue) {
166000 ...* Possible optimization: Don't send value if already correct.
af9a6 ...*/
029bb3 ...outportb(IO_PORTC_ADDRESS, controlPortValue);
b4efe6 }
83af5a
a7b7b8 static void selectBoard(int board) {
237cb8 ...SetAddress(board);
92b28f ...SetControl(CTRL_BASE ^ CTRL_ADRSEL1); .../* put board ID onto address pins */
bece59 ...ioDelay(max(DELAY_ADDRESS_SETTLE, DELAY_ADRSEL1_SETTLE)); ........../* wait */
2ba5a
485205 ...SetControl(CTRL_BASE ^ CTRL_ADRSEL1 ^ CTRL_ALE); ........../* pull ALE high */
d81486 ...ioDelay(DELAY_ALE_SETTLE); ............................................/* wait */
14af5a
797450 ...SetControl(CTRL_BASE ^ CTRL_ADRSEL1); ........../* pull ALE back */
fc1486 ...ioDelay(DELAY_ALE_SETTLE); ............................................/* wait */
36af5a
7c9619 ...SetControl(CTRL_BASE); ............................................/* ADRSEL1 done */
b11e79 ...ioDelay(DELAY_ADRSEL1_SETTLE); ............................................/* wait */
39af5a
6417d7 CURRENT_BOARD = board;
14d35d CURRENT_CHIP := -1;
c4efe6 }
5baf5a
e6af5a
```
Chapter 5: Software Source Code

```c
/*
 * reset
 * chip
 */
/*
 * board,
 * */
/*
 * board,
 * •
 * -This
 * 5-14
 */
/*
 * -pull ALE high *
 * 1c14b6 -ioDelay(DELAY_ALE_SETTLE); ................................./* wait */
faa5f9
917415 -SetControl(CTRL_BASE); .........................................../* configure the 10 port */
8baf3a
8af5a
fb6496 -CURRENT_CHIP = chip;
4fefe6 
)
d6af5a
8ba5a
d13072 void SetBaseAddress(int address) {
  eee199 -IO_BASE_ADDRESS = address;
  2fefe6 
}
34a45a
a7a5a
1a38e5 /*
2c952d -RESET A SINGLE BOARD
8a775e *
810727 -This function resets an entire board. It is not optimized for speed.
c18338 -It is necessary to delay after calling this function until the board
30b84c *-reset completes.
11495d */
6aaf7b4 int ResetBoard(int board) {
 6aaf5a
65da81 /* Configure the 10 card (doesn't matter if for data input or output) */
048b86 -ConfigureIO_Port(CNFG_INPUT); ......................................./* configure the 10 port */
3c750b -ConfigureIO_Port(CNFG_OUTPUT); ...................................../* configure the 10 port */
0daa5f 
23fbb6 -selectBoard(board); ....................................................../* select the board */
3daa5f 
356d9 -SetControl(CTRL_BASE ^ CTRL_ALE);
1d9c65 -ioDelay(DELAY_ALE_SETTLE); ......................................./* wait */
c33f1e -SetControl(CTRL_BASE);
63941c -ioDelay(DELAY_ALE_RECOVER);
09a5f9 
27a46e -CURRENT_BOARD = -1; ..................................................../* reset this on next 10 to be safe */
15e381 -CURRENT_CHIP = -1; ..................................................../* reset this to be safe */
56c89a -return (0);
52fe6 
65aaf5a
40a5a
0ca707 void SetRegister(int board, int chip, int reg, int value) {
  b637bb -if (CURRENT_PORT_CNFG != CNFG_OUTPUT) ................/* set 10 data lines for output */
  a221ba -ConfigureIO_Port(CNFG_OUTPUT);
  5277f8 -if (CURRENT_CHIP != board) .................................../* make sure board is selected */
  5ef2c3 -selectBoard(board);
  cad37b -if (CURRENT_CHIP != chip) ....................................../* make sure chip is selected */
  ae938d -selectChip(chip);
  aafa5a 
739c26 -SetAddress(reg); ....................................................../* select the right address */
  2e3549 -SetData(value); ....................................................../* output the data */
  1d9b46 -SetControl(CTRL_BASE ^ CTRL_ADRSEL2); ................./* pull low */
  a0b7f8 -ioDelay(max(DELAY_ADDRESS_SETTLE,DELAY_DATA_SETTLE),
  8a3b6 -............................DELAY_ADRSEL2_SETTLE));
  bb7938 -SetControl(CTRL_BASE ^ CTRL_WRB ^ CTRL_ADRSEL2); ........../* pull WRB low */
  7a1a1d -ioDelay(DELAY_WRB_HOLD); ......................................./* wait */
  625cfe -SetControl(CTRL_BASE ^ CTRL_ADRSEL2); ........................./* hold it */
  06b257 -ioDelay(DELAY_WRB_RECOVER); ..................................../* pull WRB high again */
  3ab463 -SetControl(CTRL_BASE); .........................................../* let WRB high again */
  ab6b54 -ioDelay(DELAY_ADRSEL2_SETTLE); ................................./* wait */
c6f5af 
2890e3 int GetRegister(int board, int chip, int reg) {
  7a6bfb -int rval;
  c6a5a
0ba150 -if (CURRENT_PORT_CNFG != CNFG_INPUT) ................/* set 10 data lines for input */
  f6b51f -ConfigureIO_Port(CNFG_INPUT);
  4277f8 -if (CURRENT_BOARD != board) ................................/* make sure board is selected */
```
5772c3 selectBoard(board);           /* make sure chip is selected */
c0d37b if (CURRENT_CHIP != chip)     
c8a15a selectChip(chip);            

5e2a826 selectBoard(board);          /* make sure chip is selected */
e823d4 SetAddress(reg);              /* select the right address */
e275ff ioDelay(max(DELAY_ADDRESS_SETTLE, DELAY_ADRSEL2_SETTLE));  /* pull AdrSel2 low */
6fe7c0 SetControl(CTRL_BASE ^ CTRL_ADRSEL2);  /* pull RDB low */
fa8603 ioDelay(DELAY_RDB_HOLD);

cbcd573 SetControl(CTRL_BASE ^ CTRL_ADRSEL2);  /* let RDB high */
c2935 ioDelay(DELAY_RDB_RECOVER);

6ada1b3 SetControl(CTRL_BASE);       /* let ADRSEL2 high */

9a80ca ioDelay(DELAY_ADRSEL2_SETTLE);

25a73a return (rval);

60e860 return (0);

7a97a8 int CheckRegister(int board, int chip, int reg, int value) {

90b207 int i = GetRegister(board, chip, reg);

C2400d if (i != value) return (-1);

15ef60 return (0);
---5810 001b3720ca780020006 Page 1 of chipio.h

8d2d03 /***************************************************************
1a43fa * chipio.h ..............................................................*
7c1e7f * ....................................................................*
52f9fa * ........................Header file for chipio.c ................*
45717f * ....................................................................*
3bc9fa * ....................................................................*
7709fc * ................................Written 1998 by Cryptography Research
0e90a3 (http://www.cryptography.com) .....................................*
7709fc * ................................and Paul Kocher for the Electronic Frontier
0e90a3 Foundation (EFF) .........................................................*
e6cae8 * ................................Placed in the public domain by
28f9fa Cryptography Research and EFF ......................................*
e6cae8 * ................................THIS IS UNSUPPORTED FREE SOFTWARE.
28f9fa USE AND DISTRIBUTE AT YOUR OWN RISK ......................*
8b4992 * ....................................................................*
1329eb * ....................................................................*
046eef * ................................IMPORTANT: U.S. LAW MAY REGULATE
2b29eb THE USE AND/OR EXPORT OF THIS PROGRAM .................*
ad28d9 * ....................................................................*
2b29eb * ................................Revision History: .................................
046eef * ....................................................................*
ad28d9 * ....................................................................*
046eef * ................................Version 1.0: Initial release by
2b29eb Cryptography Research to EFF .................................
046eef * ....................................................................*
046eef * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
1329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
046eef * ....................................................................*
2329eb * ....................................................................*
5-17

---17a 080290958180202007 Page 1 of des.c

8d2d03 /* des.c */
921603 * Software Model of ASIC DES Implementation
a8bca9 *
9b29eb * Written 1995-8 by Cryptography Research (http://www.cryptography.com)
b2c441 * THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTION AT YOUR OWN RISK.
3d3492 *
4829eb *
25c755 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM.
d19eb *
c6489b *
8229eb *
6b15cb * IMPLEMENTATION NOTES:
2b7661 *
217602 * This DES implementation adheres to the FIPS PUB 46 spec and produces ...
a5ad64 * standard output. The internal operation of the algorithm is slightly ...
bd2a2b * different from FIPS 46. For example, bit orderings are reversed ...
7d1be * the right-hand bit is now labelled as bit 0), the 5 tables have ...
bab9c7 * rearranged to simplify implementation, and several permutations have ...
e3c21e * been inverted. For simplicity and to assist with testing of hardware ...
c52d8b * implementations, code size and performance optimizations are omitted.
5d29eb *
3c489b *
1929eb *
e76eef * -REVISION HISTORY:
d229eb *
cc6443 * - Version 1.0: Initial release -- PCK.
7bb74c * - Version 1.1: Altered DecryptDES exchanges to match EncryptDES.-- PCK.
425c27 * - Version 1.2: Minor edits and beautifications.-- PCK
b3d930 * - Version 1.3: Changes and edits for EFF DES Cracker project.
ad29eb *
83d8c3 *
52af5a **
bf4eb2 #include <stdio.h>
4b3e3 #include <stdlib.h>
2e324c #include <string.h>
0a2bac #include "des.h"
32af5a static void ComputeRoundKey(bool roundKey[56], bool key[56]);
3877e0 static void RotateRoundKeyLeft(bool roundKey[56]);
9bceca static void RotateRoundKeyRight(bool roundKey[56]);
615104 static void ComputeIP(bool L[32], bool R[32], bool inBlk[64]);
3707da static void ComputeFP(bool outBlk[64], bool L[32], bool R[32]);
46017b static void ComputeF(bool out[32], bool R[32], bool roundKey[56]);
3649fe static void ComputeP(bool output[32], bool input[32]);
ab7fae static void ComputeS Lookup(int k, bool output[4], bool input[6]);
6e3b8e static void ComputePC2(bool subKey[48], bool roundKey[56]);
a3dif9c static void ComputeExpansionE(bool expandedBlock[48], bool R[32]);
baf30 static void DumpBin(char *str, bool *b, int bits);
8c4d3c static void Exchange_Land_R(bool L[32], bool R[32]);
87c6f5a 27ade5 static int EnableDumpBin = 0;
02af5a b0baf5a 00af5a c0af5a
964d6c /* **************************************************************/
dec68f /* DES TABLES */
935c1a /* */
46c68f /* */
094d6c /* **************************************************************/
3dafa5 b0af5a 7549a /*
f6556a * IP: Output bit table_DES_IP[ ] equals input bit i.
a3495d */
51c166 static int table_DES_IP[64] = {
419d69 39, -7, 47, 15, 53, 23, 63, 31,
b9c82 58, -6, 46, 14, 54, 22, 62, 30,
f838ae 37, -5, 45, 13, 53, 21, 61, 29,
5c6ed0 -36, -4, 44, 12, 52, 28, 60, 28,
868251 -33, -3, 43, 11, 51, 19, 69, 27,
29e709 -34, -2, 42, 10, 50, 18, 58, 26,
/* FP: Output bit table_DES_FP[i] equals input bit i. */
eadd2a static int table_DES_FP[64] = {
  25, 47, 49, 33, 25, 17, 9, 1, 
  59, 51, 43, 35, 27, 19, 11, 3, 
  61, 53, 45, 37, 29, 21, 13, 5, 
  63, 55, 47, 39, 31, 23, 15, 7, 
  56, 48, 40, 32, 24, 16, 8, 9, 
  58, 50, 42, 34, 26, 18, 10, 2, 
  60, 52, 44, 36, 28, 20, 12, 4, 
  62, 54, 46, 38, 30, 22, 14, 6 
};

5af5a
20af5a

838e5 /*
  3cd085 /* PC1: Permutation choice 1, used to pre-process the key
  0495d */
c00c38 static int table_DES_PC1[56] = {
  27, 19, 11, 31, 39, 47, 55, 
  26, 18, 10, 38, 46, 54, 
  29, 17, 9, 29, 47, 53, 
  24, 16, 8, 28, 36, 52, 
  23, 15, 7, 35, 43, 51, 
  22, 14, 6, 34, 42, 50, 
  21, 13, 5, 33, 41, 49, 
  20, 12, 4, 32, 40, 48 
};
b08f7f 

55af5a
dfa5a

8438e5 /*
  45f37a /* PC2: Map 56-bit round key to a 48-bit subkey
  0495d */
067fcf static int table_DES_PC2[48] = {
  24, 27, 20, 6, 14, 10, 3, 22, 
  9, 17, 12, 8, 23, 11, 5, 
  16, 26, 11, 9, 19, 25, 4, 15, 
  56, 43, 36, 29, 49, 40, 48, 30, 
  52, 44, 37, 33, 46, 35, 50, 41, 
  28, 53, 51, 55, 32, 45, 39, 42 
};

a82f7f 

55af5a
dfa5a

838e5 /*
  1cd3f1 /* E: Expand 32-bit R to 48 bits.
  76495d */
ac6a87 static int table_DES_E[48] = {
  31, 0, 1, 2, 3, 4, 3, 4, 
  5, 6, 7, 8, 9, 10, 
  11, 12, 13, 14, 15, 16, 
  15, 16, 17, 18, 19, 20, 
  21, 22, 23, 24, 25, 26, 
  27, 28, 29, 30, 31, 0 
};

482f7f 

5af5a
dfa5a

365137 static int table_DES_PC3[32] = {
  11, 17, 5, 27, 25, 10, 20, 0, 
  15, 21, 3, 28, 29, 7, 18, 24, 
  31, 22, 6, 26, 2, 16, 8, 
  14, 30, 4, 19, 1, 9, 15, 23 
};

eaf5a
20af5a
Chapter 5: Software Source Code

5-19

--f1a0 00f7378b1880020007 Page 3 of des.c

bc38e5 */
8e6f34 */ S Tables: Introduce nonlinearity and avalanche
a3495d */
a11e19 static int table_DES[58][64] = {
3cd69a /* table S[0] */
e4846d /* table S[1] */
c965fa /* table S[2] */
95d5b0 /* table S[3] */
e8c3e1 /* table S[4] */
11d2b1 /* table S[5] */
d54b6a /* table S[6] */
98d4fd /* table S[7] */
35f575 /* table S[8] */
7baf5a /* table S[9] */
d2dec /* table S[10] */
9bd582 /* table S[12] */
7f254f /* table S[13] */
9a7505 /* table S[14] */
6bdae7 /* table S[15] */
385c4d /* table S[16] */
37f5dc /* table S[17] */
d3ed33 /* table S[18] */
57fc36 /* table S[19] */
2e6b58 /* table S[20] */
45f43f /* table S[21] */
9ed830 /* table S[22] */
49c2de /* table S[23] */
bb1e9b /* table S[24] */
03bd2c /* table S[25] */
bed567 /* table S[26] */
3bb261 /* table S[27] */
49c2d0 /* table S[28] */
69a636 /* table S[29] */
38ac1c /* table S[30] */
6b7311 /* table S[31] */
aeffa /* table S[32] */
ceca4b /* table S[33] */
68c766 /* table S[34] */
7baac0 /* table S[35] */
40c7ed0 /* table S[36] */
31f777 /* table S[37] */
18812f7;}
34a5f
53a5fa e6af5a
fcad5a
d34d6c /******************************************************************************/
d34d6c /*DES CODE*****************************************************************************/
fbca5f /*------------------------------*/
ffcd685f /*------------------------------*/
47c46d /******************************************************************************/
8eaf5a
5ba5fa 2a38e5 */ EncryptDES: Encrypt a block using DES. Set verbose for debugging info.
47e70b /* (This loop does both loops on the "DES Encryption" page of the flowchart.)
da5620 void EncryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose) {
06bb1c /* Int f round;*/
9b92a5 /* bool RC[32]; L[32], fout[32];*/
6bfaf /* bool roundKey[56];*/
d8af5a 9b1294 /* EnableDumpBin = verbose;*/
3cc2a /* DumpBin("input(left)", inBlk+32, 32);*/
7281b2 /* DumpBin("input(right)", inBlk, 32);*/
490a8e /* DumpBin("raw key(left )", key+28, 28);*/
d75585 /* DumpBin("raw key(right)", key, 28);*/
ffca5a f7c1be /* Compute the first roundkey by performing PC1 */
Chapter 5: Software Source Code

---1527 00053c1ed1980020007 Page 4 of des.c

47b264: .ComputeRoundKey(roundKey, key);
3fa5a: 0295d4: .DumpBin("roundKey(L)", roundKey+28, 28);
840af5: .DumpBin("roundKey(R)", roundKey, 28);
4baa5a: /* Compute the initial permutation and divide the result into L and R */
1a1a8: .ComputeIP(L,R,inBlk);
aea5a: 7777ba: .DumpBin("after IP(L)", L, 32);
cc7699: .DumpBin("after IP(R)", R, 32);
ba7f5a: 1f6637: for (round = 0; round < 16; round++) {
4221bf: if (verbose) {
4a9a10: .printf("-------- BEGIN ENCRYPT ROUND %d -----------\n", round);
838034: .DumpBin("round start(L)", L, 32);
628117: .DumpBin("round start(R)", R, 32);
d2af5a: 8b7f1c: /* Rotate roundKey halves left once or twice (depending on round) */
aec8ba: .RotateRoundKeyLeft(roundKey);
9b1467: if (round != 0 && round != 1 && round != 8 && round != 15) 1950e7: .RotateRoundKeyLeft(roundKey);
003c8d: .DumpBin("roundKey(L)", roundKey+28, 28);
b1bd4: .DumpBin("roundKey(R)", roundKey, 28);
82af5a: e0033b: /* Compute f(R, roundKey) and exclusive-OR onto the value in L */
14b969: .ComputeF(fout, R, roundKey);
5c5a47: .DumpBin("f(R,key)", fout, 32);
4a4739: for (i = 0; i < 32; i++)
2a9e6: .LL[i] ^= fout[i];
a5a5ab: .DumpBin("L^f(R,key)", L, 32);
5aaf5a: 7a68b4: .Exchange_L.and_R(L,R);
3af5a: 3a3f40: .DumpBin("round end(L)", L, 32);
5ca663: .DumpBin("round end(R)", R, 32);
4b21bf: if (verbose) {
a2451a: .printf("-------- END ROUND %d -----------\n", round);
66df1c: };
9af5a: 5e8e68: .Exchange_L.and_R(L,R);
87af5a: 71370b: /* Combine L and R then compute the final permutation */
77cf94: .ComputeF(outBlk, L,R);
46b991: .DumpBin("FP out(left)", outBlk+32, 32);
cbf675: .DumpBin("FP out(right)", outBlk, 32);
b8ffe6: 43af5a: 15af5a: 01af5a: e038e5: /* DecryptDES: Decrypt a block using DES. Set verbose for debugging info.
6fa5c7: /* (This loop does both loops on the "DES Decryption" page of the flowchart.)
1a495d: */
b66d58: void DecryptDES(void key[56], void outBlk[64], void inBlk[64], int verbose) {
8b2bc: .int i,round;
2999a1: .bool R[32], L[32], fout[32];
f9bfaf: .bool roundKey[56];
24af5a: 5a1294: .EnableDumpBin = verbose; ....................... /* set debugging on/off flag */
96cbb2: .DumpBin("input(left)", inBlk+32, 32);
4b8fb2: .DumpBin("input(right)", inBlk, 32);
f608be: .DumpBin("raw key(left)", key+28, 28);
395585: .DumpBin("raw key(right)", key, 28);
82af5a: 8c11be: /* Compute the first roundkey by performing PC1 */
46b264: .ComputeRoundKey(roundKey, key);
dcafa5: 5f195d: .DumpBin("roundKey(L)", roundKey+28, 28);
30aaf5: .DumpBin("roundKey(R)", roundKey, 28);
35aaf5a: 291340: /* Compute the initial permutation and divide the result into L and R */
53d1a8: .ComputeIP(L,R,inBlk);
```
1f01 0005e2de84280020007 Page 5 of des.c

```
Chapter 5: Software Source Code

--1bb7 0002e9cf80d80020007 Page 6 of des.c

e0af5a 5-22 0x0

d38e85 /* RotateRoundKeyRight: Rotate each of the halves of roundKey right one bit */
f9495d */

1ab26c static void RotateRoundKeyRight(bool roundKey[56]) {
    7d483e ... bool temp1, temp2;
    1517e0 ... int i;
    c0af5a 7a5025 ... temp1 = roundKey[0];
    9c6548 ... temp2 = roundKey[28];
    5fe508 ... for (i = 0; i < 27; i++) {
196cc2 ... .roundKey[i] = roundKey[i+1];
    789d9a ... roundKey[i+28] = roundKey[i+28+1];
    ecdflc ... }
    dfa88d ... roundKey[27] = temp1;
    945d11 ... roundKey[55] = temp2;
    e3e6f6 }
2c6f5a 41af5a 38af5a 7738e5 /* ComputeIP: Compute the initial permutation and split into L and R halves. */
d4f59d */

23ac44 static void ComputeIP(bool L[32], bool R[32], bool inBlk[64]) {
    686085 ... bool output[64];
    d417e0 ... int i;
    a3af5a 77aeaf ... /* Permute */
07f9a6 ... */

4e6406 ... for (i = 63; i >= 0; i--)
adc750 ... output[IPtable[i]] = inBlk[i];
95af5a 500318 ... /* Split into R and L. Bits 63..32 go in R, bits 31..0 go in R. */
70f9a6 ... */

daba85 ... for (i = 63; i >= 0; i--)
40368 ... ... if (i >= 32)
3ef2b8 ... ... L[i-32] = output[i];
6b70b5 ... ... else
1b708d ... ... R[i] = output[i];
86df1c ... }

f5ef6e }
4baf5a 70af5a 2aaf5a 3738e5 /* ComputeFP: Combine the L and R halves and do the final permutation. */
84efef */

d1af55 static void ComputeFP(bool outBlk[64], bool L[32], bool R[32]) {
34e93e ... bool input[64];
5c17e0 ... int i;
08af5a 2ac6c1 ... /* Combine L and R into input[64] */
45f9a6 ... */

916406 ... for (i = 63; i >= 0; i--)
858397 ... ... input[i] = (i >= 32) ? L[i - 32] : R[i];
8caf5a d3aeaf ... /* Permute */
98f9a6 ... */
1c6406 ... for (i = 63; i >= 0; i--)
85e116 ... ... outBlk[IPtableFPCi] = input[i];
bec6fe }
18af5a befaf5a ... /* Expand R into 48 bits using the E expansion */
Chapter 5: Software Source Code

--92d2 000e8f1171f80020007 Page 7 of des.c

fb90d7:  *ComputeExpansionE(expandedBlock, R);
e5f0ba:  *DumpBinC("expanded E", expandedBlock, 48);
efa5fa:  7693ff:  /* Convert the roundKey into the subkey using PC2 */
4a7840:  *ComputePC2(subkey, roundKey);
21d717:  *DumpBinC("subkey", subkey, 48);
39af5a:  e5154c:  /* XOR the subkey onto the expanded block */
bffcab:  *for (i = 0; i < 48; i++)
2d6512:  *computedBlock[i] ^= subkey[i];
fb5f5a:  390740:  /* Divide expandedBlock into 6-bit chunks and do S table lookups */
0225c6:  *for (k = 0; k < 8; k++)
e985c7:  *computeS_Lookup(k, sout+4*k, expandedBlock+6*k);
67af5a:  dbfd35:  /* To complete the f() calculation, do permutation P on the S table output */
6d2d52:  *ComputeP(fout, sout);
b3efe6:  
0ca5f5a:  76af5a
d3af5a:  5338e5:  /*
eb913f:  * *ComputeP: Compute the P permutation on the S table outputs.
52495d:  */
61f410:  static void ComputeP(bool output[32], bool input[32]) {
3b17e0:  *int i;
20af5a:  5339a:  *for (i = 0; i < 32; i++)
3f7688:  *output[(table_DES_PC2[i] = input[i];
b1efe6:  10af5a:  9af5a
78af5a:  d3af5a:  5338e5:  /*
8a859b:  * Look up a 6-bit input in S table k and store the result as a 4-bit output.
ad495d:  */
49a67e:  static void ComputeS_Lookup(int k, bool output[4], bool input[6]) {
9bf3da:  *int inputValue, outputValue;
a5af5a:  ad19e:  /* Convert the input bits into an integer */
f18c1a:  *------16*input[4] + 32*input[5];
3fa5fa:  164a3e:  /* Do the S table lookup */
34b706:  *outputValue = table_DES_S[k][inputValue];
4baf5a:  0a8aee:  /* Convert the result into binary form */
0c9a60:  *output[0] = (outputValue & 1) ? 1 : 0;
3ab6ec:  *output[1] = (outputValue & 2) ? 1 : 0;
d6a848:  *output[2] = (outputValue & 4) ? 1 : 0;
708c7f:  *output[3] = (outputValue & 8) ? 1 : 0;
95efe6:  
dfa5fa:  78af5a
d2af5a:  0338e5:  /* *ComputePC2: Map a 56-bit round key onto a 48-bit subkey
8181cc:  */
2a495d:  */
22796f:  static void ComputePC2(bool subkey[48], bool roundKey[56]) {
2817e0:  *int i;
d0af5a:  17fcb:  *for (i = 0; i < 48; i++)
bec8bc:  *subkey[i] = roundKey[(table_DES_PC2[i])];
16efe6:  67af5a
c5af5a:  43af5a:
2838e5:  /*
28459d:  */
4bb46d:  static void ComputeExpansionE(bool expandedBlock[48], bool R[32]) {
e817e0:  *int i;
```c
for (i = 0; i < 48; i++)
expandedBlock[i] = Rtable[DES_E[i]];
```

```c
static void Exchange_L_and_R(bool L[32], bool R[32]) {

for (i = 0; i < 32; i++)
LC[i] = RC[i];
```
```c
#ifndef _DES_H
#define _DES_H
typedef char bool;

void EncryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);
void DecryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);
#endif
```
Chapter 5: Software Source Code

5-27

--cfc3 000b6391f9f8020009 Page 2 of initsrc.h

87048b ....... printf("Parameters can be entered on the command line or entered ",
a20e7c "manually.\n\nUsage modes: \(ctx=\text{ciphertext, ptx=plaintext}\)\n084ab3 "\(\text{desbrute search.ctxt K (8 bytes ptxt) (8 bytes ctext)}\)!\n3c43c "\(\text{desbrute search.ctxt E (8 bytes ctext0) (8 bytes ctext1)}\)!\n19241a "\(\text{desbrute search.ctxt C (8 bytes IV) (8 bytes ctext0)}\)"
2d89d0 "\(\text{\(8 \text{bytes ctext1)}\)!\n3d83b5 "\(\text{desbrute search.ctxt B (1 ctext byte to repeat)}\)!\n42dab1 "\(\text{desbrute search.ctxt M (ptxtVec) (IV) (cxt0) (cxt1)}\)"
71870 evt & \(\text{dMask) (schint)}\)!\n2cf69b "\(\text{Parameters can also be input from a file (e.g., "}
c31498 "\text{\textfiles\"})\n36df1c ")
41tf5a
8b33db4 
-*** OPEN OUTPUT FILE ****/
142284 if (argc > nextArg) {
b3d4d ... c = argv[nextArg+1];
5449d8 3da6db "\text{printf("Enter output file for search context [ENTER=\"search.ctxt\"]:");}
7e0aa5 7a5331 if (*buf == '\0')
6ea08e "\text{strcpy(buf, \"search.ctxt\");}
383b52 c = buf;
2bdf1c 

e2744c 
-*** INITIALIZE searchType ****/
e92284 if (argc > nextArg) {
053ed4 ... c = argv[nextArg+1];
7949d8 else {
1c6455 "\text{printf("The array supports a variety of search types:\n")};
893c39 "\text{printf("K - Known plaintext (standard brute force)\n")};
453836 "\text{printf("E - ECB ASCII text\n")};
ebe877 "\text{printf("C - CBC ASCII text\n")};
c042f4 "\text{printf("B - Blaze challenge\n")};
f72546 "\text{printf("M - Manual parameter specification\n")};
457820 "\text{printf("Enter search type: ");}
2e98f5 "\text{fgets(buf, 99, stdin);}
6f3b52 ... c = buf;
16df1c 
\text{bde55a} 
-*** INITIALIZE PARAMETERS FOR KNOWN PLAINTEXT SEARCHES ****/
b606fd if (searchType == \'K\') {
40a51a 
31b0ef 
-# Get known plaintext #
b6f875 if (argc > nextArg) {
66d208 ... c = argv[nextArg+1];
2a89a9 else {
865acf "\text{printf("Enter known plaintext (16 hex digits): ");}
44e536 "\text{fgets(buf, 99, stdin);}
519eeec c = buf;
71e7fe 
3b268c if (unhex(plaintext, c, 8))
5284f6 "\text{EXIT_ERR("Invalid plaintext. (Must be 16 hex digits)\")};
43a5a d154a2 /* Get ciphertext 0 (use same for ciphertext 1) */
99f875 if (argc > nextArg) {
b6d208 ... c = argv[nextArg+1];
286a79 else {
116333 "\text{printf("Enter ciphertext (16 hex digits): ");}
ebec56 "\text{fgets(buf, 99, stdin);}
d79eeec c = buf;
386f7e 
\text{a0e9c} if (unhex(ctx.ciphertext0, c, 8) || unhex(ctx.ciphertext1, c, 8))
ed501c "\text{EXIT_ERR("Invalid ciphertext. (Must be 16 hex digits)\")};
0ba5a 95b74c /* Set ctx */
b6b998 "\text{memset(ctx.plaintextVector, 0, size(ctx.plaintextVector));}
Chapter 5: Software Source Code

---8f53 000bd86da9880020009 Page 3 of initsrch.c

b1371 ::for (i = 0; i < 8; i++)
0c2965 ::ctx.plaintextVector[plaintext[i]/8] = (1 << (plaintext[i] % 8));
9d69b5 ::ctx.plaintextByteMask = 0x00;
23175f ::memset(ctx.plaintextXorMask, 0, sizeof(ctx.plaintextXorMask));
d8f16c ::ctx.searchinfo = 16;
0f3f1c ::/* useCBC=0, extraXor=0, boardActiveEn=1 */
10a5fa ::
1aad39 ::/** INITIALIZATE PARAMETERS FOR ASCII SEARCHES ***/
0a4571 ::if (searchType == 'E' || searchType == 'C') {
10d9b2a ::/* Get IV (only if this is ciphertext mode) */
2d53b8 ::if (argc > nextArg) {
0a4e47 ::if (argc > nextArg) {
1a9a66a ::c = argv[nextArg+1];
dbcfc7 ::} else {
0493c ::printf("Enter IV (16 hex digits): ");
653c62 ::fgets(buf, 99, stdin);
20cc35 ::c = buf;
8f42cc ::}
177471 ::if (unhex(ctx.plaintextXorMask, c, 8))
906bc1 ::EXIT_ERR("Invalid IV. (Must be 16 hex digits.");
e26fe7 ::
30a5fa ::
16b543 ::* Get ciphertext 0 */
8c1875 ::if (argc > nextArg) {
0d2080 ::c = argv[nextArg+1];
b86a79 ::} else {
1c808a ::printf("Enter ciphertext0 (16 hex digits): ");
e2ec56 ::fgets(buf, 99, stdin);
819ee9 ::c = buf;
636fe7 ::}

d01f3c8 ::if (unhex(ctx.ciphertext0, c, 8))
ab36df ::EXIT_ERR("Invalid ciphertext0. (Must be 16 hex digits.");
4afa59 ::
83be07 ::* Get ciphertext 1 */
2d1875 ::if (argc > nextArg) {
7a0d280 ::c = argv[nextArg+1];
66a79 ::} else {
303849 ::printf("Enter ciphertext1 (16 hex digits): ");
6cc56 ::fgets(buf, 99, stdin);
89eeec ::c = buf;
e66fe7 ::}

54dfe9 ::if (unhex(ctx.ciphertext1, c, 8))
8b3ebe ::EXIT_ERR("Invalid ciphertext1. (Must be 16 hex digits.");
82af5a ::
14b74c ::* Set ctx */
37b998 ::memset(ctx.plaintextVector, 0, sizeof(ctx.plaintextVector));
66346e ::for (i = 0; i < sizeof(asciiBytes); i++)
ff7edd ::ctx.plaintextVector[casciBytes[i]/8] = (1 << (asciiBytes[i] % 8));
760bb5 ::ctx.plaintextByteMask = 0x00;
a8bf8c ::if (searchType == 'E') {
c97dd1 ::memset(ctx.plaintextXorMask, 0, sizeof(ctx.plaintextXorMask));
bc0a8e ::ctx.searchinfo = 16; /* useCBC=0, extraXor=0, boardActiveEn=1 */
476792 ::} else {
60ba9b ::* already set plaintextXorMask = IV */
61c50 ::ctx.searchinfo = 17; /* useCBC=1, extraXor=0, boardActiveEn=1 */
706fe7 ::
4bdf1c ::}

1baf5a ::
e0b98e ::/** INITIALIZATE PARAMETERS FOR BLAZE CHALLENGE ***/
9f2b38 ::if (searchType == 'B') {
6ca5fa ::
207380 ::* Get ciphertext byte */
c6b1875 ::if (argc > nextArg) {
8d2080 ::c = argv[nextArg+1];
266a79 ::} else {
1db8e7 ::printf("Enter ciphertext byte (2 hex digits): ");
cfc65 ::fgets(buf, 99, stdin);
899eeec ::c = buf;
2c6fe7 ::}
79aa0b ::if (unhex(ctx.ciphertext0, c, 1))
b2c25b ::EXIT_ERR("Invalid ciphertext byte. (Must be 2 hex digits.");

/* Initialize parameters for manual mode */

/* Get plaintextVector */
if (argc > nextArg)
    c = argv[nextArg++];

/* Get ciphertext0 */
if (argc > nextArg)
    c = argv[nextArg++];

/* Get ciphertext1 */
if (argc > nextArg)
    c = argv[nextArg++];

/* Get ciphertext0 and ciphertext1 bytes to the input byte */
...
Chapter 5: Software Source Code

---7835 00008e1996880020009 Page 5 of initsrch.c

e2a994 ....printf("be set to 1. For example, if the left-hand plaintext byte
")
0c1884 ....printf("is unknown, the mask would be 0x80\n\n\n")
35f45f ....printf("Enter plaintextByteMask (1 byte): ");
5f863b ....fopen(buf, 99, stdin);
e599ec ....c = buf;
5c6f67 ....)
1d19bc ....if (unhex((ctx.plaintextByteMask), c, 1))
0603d8 ....EXIT_ERR("Invalid plaintextByteMask. (Must be 2 hex digits.");
d1af5a
9cf875 ....if (argc > nextArg) {
03d208 ....c = argv[nextArg++];
c46a79 ....} else {
60efb7 ....printf("\n\nThe searchInfo byte has two search parameters: \n\n")
5f873b ....printf("- bit 0x10: boardActiveEnable. Set to one.\n")
6137c1 ....printf("- bit 0x02: extraXor. If set, after the decryption is done, \n")
6555ba ....printf("- the right half is XORed onto the left. \n")
2b6a7f ....printf("- This is for Matt Blaze's challenge. \n")
7931e2 ....printf("- bit 0x01: useCBC. If set, the first ciphertext is XORed \n")
1779ee ....printf("- onto the second plaintext before the second\n")
7a8401 ....printf("- plaintext is checked against the ");
c4a0b6 ....printf("plaintextVector. (Higher bits control)\n");
5295d9 ....printf("searchActive, which is currently unused. \n")
411e25 ....printf("Enter searchInfo (1 byte): ");
6ee5c6 ....fgets(buf, 99, stdin);
859ee ....c = buf;
a16fe7 ....}
57ac83 ....if (unhex((ctx.searchInfo), c, 1))
196157 ....EXIT_ERR("Invalid searchInfo. (Must be 2 hex digits.");
a5df1c ....
42a05a
73761b ....printf("\n\n\nSEARCH PARAMETERS ");
630d8c ....printf("\n\n\n");
e6f965 ....dumpBin("- ptxVector = ", ctx.plaintextVector, 32);
22c115 ....dumpBin("- ptxXorMask = ", ctx.plaintextXorMask, 8);
dcbce6 ....dumpBin("- ciphertext0 = ", ctx.ciphertext0, 8);
51ed2d ....dumpBin("- ciphertext1 = ", ctx.ciphertext1, 8);
48bf8f ....dumpBin("- ciphertextMask = ", &ctx.plaintextByteMask, 1);
8f310d ....dumpBin("- searchInfo = ", &ctx.searchInfo, 1);
5a4a25 ....printf("\n\n\n\n\n");
90f47f ....printf("\n\n\n");
dfa5a
022e67 ....\**** WRITE SEARCH PARAMETERS TO OUTPUT FILE ****/
6eb50f ....printf("\n\nWriting output file...\n");
14b77f ....fflush(stdout);
1a556e ....WriteSearchContext(outfile, &ctx);
1655cf ....fclose(outfile);
48b12f ....printf("Done.\n");
2bc86a ....return (0);
fae6f6 ....
76a5a
03a5a
5e3e65 /* ....Print a descriptive string followed by a binary value (in hex)
42a495d */
15e54b static void dumpBin(char *intro, unsigned char *data, int len) {
1d17e0 ....int i;
313b28 ....printf(intro);
91c199 ....for (i=len-1; i >= 0; i--)
1b8cd7 ....printf("\%02X", data[i]);
799ee8 ....printf("\n");
1ce8ed ....}
c8a5f6 
5a5a5a
2daf5a
9f58e5 /* ....Convert an ASCII digit from hex to an int, or return -1 if not hex.
6a9929 */
a55514 static int unhexDigit(char c) {
4253c4 ....if (c >= '0' && c <= '9')
5cb6d6 ....return (c - '0');
5db5b1 ....if (c >= 'A' && c <= 'F')
3b8bda ....return (c - 'A' + 10);
0849e3 ....if (c >= 'A' && c <= 'F')

--2b51 001fdd8d50d80020009 Page 6 of initsrch.c

0da66f 0... return (c - 'A' + 10);
59a5eb 0... return (-1); ............................ /* return -1 for error: bad hex digit */
8def6 1cfa5a
5eaf5a
2238e5 /* 1d22f5 */ . Convert a string of hex characters into unsigned chars.
84495d /*
62afe2 static int unhex(unsigned char *data, char *hex, int byteCount) {
01e13a 0... int i,j;
6aaf5a
e6aae1 0... if (data == NULL || hex == NULL)
128f8d 0... return(-1);
afa5a
852a60 /* Remove comments and whitespace */
97d4c3 0... for (i=j=0; hex[i] != 0 && hex[i] != '%' && hex[i] != '#'; i++)
7f01aa 0... if (hex[i] == ' ')
7c72f3 0... hex[i+j] = hex[i];
3bf89c 0... hex[j] = '\0';
a1af5a
665c18 0... if ((int)strlen(hex) != byteCount*2)
3400d 0... return (-1);)
82ed9d 0... memset(data, 0, byteCount);
759629 0... for (; i < 2*byteCount; i++) {
36eb9c 0... j = unhexDigit(hex[i]);
6e001d 0... if (j < 0)
47bb9 0... return (-1);
b5824 0... data[byteCount - 1 - i/2] |= j << (((i & 1) ? 0 : 4);
66df1c 0...}
17f1eb 0... for (i = 2*byteCount; i < (int)strlen(hex); i++)
9e21eb 0... if (!isspace(hex[i]))
1cbb9 0... return (-1);
23c86a 0... return (0);
8def6 fca5a
baaf5a
fda5a
--c219 0008a71b222802000a Page 1 of keyblock.c

8d2d03 /* ***************************************************************************/
33ef1 * keyblock.c ------------------- Key Block & Search Context Management Functions ....
33ec57 * --------------- Key Block & Search Context Management Functions ..........
1c29eb * - Written 1998 by Cryptography Research (http://www.cryptography.com) ....
bf09fc * - Written 1998 by Cryptography Research (http://www.cryptography.com) ....
d58aa9 * - and Paul Kocher for the Electronic Frontier Foundation (EFF). ....
36caeb * - Placed in the public domain by Cryptography Research and EFF. ....
395992 * - THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK.
029eb* * - Version 1.0: Initial release by Cryptography Research to EFF. ....
0b28d9 * - Version 1.0: Initial release by Cryptography Research to EFF. ....
4929eb * - Version 1.0: Initial release by Cryptography Research to EFF. ....
b5d8c3 * - Version 1.0: Initial release by Cryptography Research to EFF. ....

60af5a * * ***************************************************************************/
56fe2b #include <stdio.h>
a1bea3 #include <stdlib.h>
9be465 #include <conio.h>
7632c #include <string.h>
e1c757 #include <memory.h>
8f0a5b #include <time.h>
88b1cb #include <ctype.h>
28c94c "search.h"

f92b9a static void WriteParams(FILE *fp, SEARCH_CTX *ctx);

83af5a static void ReadParams(FILE *fp, SEARCH_CTX *ctx);

3938e5 /* * Create a new search context file from a SEARCH_CTX structure

54495d */

1d4171 #define CTX_FILE_KEYBLOCKS_OFFSET (sizeof(fileHeader) + 58)
1391d3 #define MAX_KEY_REGION (1<<24) "/** 2^36 keys / 2^32 keys per region */

b5708e static void EXIT_ERR(char *s) { fprintf(stderr, s); exit(1); }

893b5f static void WriteParams(FILE *fp, SEARCH_CTX *ctx);
9000a6 static void ReadParams(FILE *fp, SEARCH_CTX *ctx);

73af5a /* * Read search params from a FILE_STRUCTURE and get ready for

5c46c * calls to ReserveKeyRegion and FinishKeyRegion.

72c385 */

f292e2 /* * Read search params from a FILE_STRUCTURE and get ready for

039c06 */

e0495d */

4ab92d void OpenSearchContext(FILE *fp, SEARCH_CTX *ctx) {

70180e long blocksLeft, n;

c17e0 } int i;

d0649a } int c;

fda5f5 da1d80 rewind(fp);

ae0b3f for (i = 0; i < sizeof(fileHeader); i++)

489af8 if (fgetc(fp) != fileHeader[1])

7c9d45 EXIT_ERR("Bad file header in search context file.

42a5f5 e43738 ReadParams(fp, ctx);

8a8edc if (ftell(fp) != CTX_FILE_KEYBLOCKS_OFFSET)
5346c8 ...EXIT_ERR("Internal error: File length mismatch.");
52a5f5a 43a1af .../ INITIALIZE THE SEARCH PROCESS PARAMETERS (except for totalUnits) */
3f0522 ...ctx->nextUnstartedKeyBlock = 0;
38d2c7 ...ctx->totalUnstartedKeyBlocks = 0;
7b9e5 ...ctx->totalUnfinishedKeyBlocks = MAX_KEY_REGION;
36d36a ...ctx->totalPendingKeyBlocks = 0;
c97c60 .../* FIND OUT HOW MANY KEY BLOCKS ARE LEFT */
e65f7b ...blocksLeft = 0;
0e785b ...for (n = 0; n < MAX_KEY_REGION/8; n++) {
3dee98 ...c = fgetc(fp);
6c3b7b ...if (ctx < 0 | | c > 255)
0b3a7e ...EXIT_ERR("Error or premature EOF reading search context file.\n");
2ab9b9 ...blocksLeft += ((c&128)/128 + (c&64)/64 + (c&32)/32 + (c&16)/16 +
e1804b ...- - - - (c&8)/8 + (c&4)/4 + (c&2)/2 + (c&1);
4091fd ...9f5ea0 ...ctx->totalUnstartedKeyBlocks = blocksLeft;
ad7c26 ...ctx->totalFinishedKeyBlocks = MAX_KEY_REGION - blocksLeft;
4defe6 }
76af5a 0aa5f5a f03a5 /*
853511 ... Reserve a key region to search. When done searching it, the program
5b54f7 ... should call FinishKeyRegion. This function hands out blocks sequentially,
468a86 ... starting with the first unsearched one in the file context file.
6a44cb ... If all blocks have been allocated and no free ones are left, the
424769 ... function returns (-1).
9c495d ...
f9dcfa long ReserveKeyRegion(FILE *fp, SEARCH_CTX *ctx) {
5c6b50 ... int c,b;
43a5f5a 824e7 ...if (ctx->nextUnstartedKeyBlock >= MAX_KEY_REGION)
e6878d ... return(-1);
d63e3f ...if (fseek(fp,CTX_FILE_KEYBLOCKS_OFFSET + ctx->nextUnstartedKeyBlock/8,
f96018 ...SEEK_SET))
276ed9 ...EXIT_ERR("Error seeking search context file.\n");
22d482 ...if (ctx->nextUnstartedKeyBlock & 7) != 0)
e19be8 ...c = fgetc(fp);
a7c101 ...while (ctx->nextUnstartedKeyBlock < MAX_KEY_REGION) {
f208b3 ...b = (int)(ctx->nextUnstartedKeyBlock & 7);
e956ec ...} if (b == 0)
40d9d8 ...c = fgetc(fp);
9a373b ...if (c < 0 | | c > 255)
d5f63d ...EXIT_ERR("Error reading from search context file.\n");
d20bf6 ...if (b == 0 & & c == 0) {
e497e7 ...ctx->nextUnstartedKeyBlock += 8;
d856fa ...continue;
7e6fe7 ...}
7e9622 ...if ((c < b) & 128)
5d88e1 ...break;
7ccac9 ...ctx->nextUnstartedKeyBlock++;
d5d0f1c ...
7842e7 ...if (ctx->nextUnstartedKeyBlock >= MAX_KEY_REGION)
2d400d ... return (-1);
204784 ...ctx->totalUnstartedKeyBlocks--;
307db7 ...ctx->totalPendingKeyBlocks++;
e6b05c ...return (ctx->nextUnstartedKeyBlock++);
daeef6 }
caaa5a 
ada5f5a 1e3a5 /*
45d7ed1 ... Finish searching a key region by marking it as completed in the context
156197 ... file. */
33495d ...
4d1ada3 void FinishKeyRegion(FILE *fp, SEARCH_CTX *ctx, long keyRegion) {
3e6b50 ... int c,b;
74af5a 04227a ...if (keyRegion < 0 | | keyRegion > MAX_KEY_REGION)
859ca72 ...EXIT_ERR("Bad key region\n");
4a7978 ...if (fseek(fp,CTX_FILE_KEYBLOCKS_OFFSET + keyRegion/8,SEEK_SET))
8799f2 ...EXIT_ERR("Error seeking in search context file.\n");
```c
/*
 * b = (int)(keyRegion & 7);
 * 
 */

b = getc(fp);

if (((c << b) & 128) == 0)
    printf("WARNING: FinishKeyRegion called, but region already searched!\n");

else {

b = (int)keyRegion / 8;

int c = getc(fp);

if ((c << b) & 128) == 0)
    printf("WARNING: FinishKeyRegion called, but region already searched!\n");

else {

fseek(fp, CTX_FILEREKEYBLOCKS_OFFSET + keyRegion/8, SEEK_SET));

EXIT_ERR("Error seeking in search context file.\n");

fputc(c & (255 ^ (128 >> b)), fp);

fflush(fp);

ctx->totalFinishedKeyBlocks++;

ctx->totalPendingKeyBlocks--;

Write a SEARCH_CTX structure to a FILE*

WriteParams(FILE *fp, SEARCH_CTX *ctx) {

fwrite(ctx->plaintextVector, 1, 32, fp); /* 32 bytes */

fwrite(ctx->plaintextXorMask, 1, 8, fp); /* 8 bytes */

fwrite(ctx->ciphertext0, 1, 8, fp); /* 8 bytes */

fwrite(ctx->ciphertext1, 1, 8, fp); /* 8 bytes */

fwrite(&(ctx->plaintextByteMask), 1, 1, fp); /* 1 byte */

fwrite(&(ctx->searchInfo), 1, 1, fp); /* 1 byte */

}

Read a SEARCH_CTX structure from a FILE*

ReadParams(FILE *fp, SEARCH_CTX *ctx) {

read(ctx->plaintextVector, 1, 32, fp); /* 32 bytes */

read(ctx->plaintextXorMask, 1, 8, fp); /* 8 bytes */

read(ctx->ciphertext0, 1, 8, fp); /* 8 bytes */

read(ctx->ciphertext1, 1, 8, fp); /* 8 bytes */

read(&(ctx->plaintextByteMask), 1, 1, fp); /* 1 byte */

read(&(ctx->searchInfo), 1, 1, fp); /* 1 byte */

}
```c
/* keyblock.h  -  Header file for keyblock.c  */

#define _KEYBLOCK_H

void WriteSearchContext(FILE *fp, SEARCH_CTX *sp);
void OpenSearchContext(FILE *fp, SEARCH_CTX *ctx);
long ReserveKeyRegion(FILE *fp, SEARCH_CTX *ctx);
void FinishKeyRegion(FILE *fp, SEARCH_CTX *ctx, long keyRegion);

#endif
```

---

**Chapter 5: Software Source Code**

5-35
Chapter 5: Software Source Code

-5629 0005fb2aed48002000c Page 1 of search.c

8d2d03 /***********************************************************/
e284a4 /* search.c */
8540b6 /* * Search Engine Controller Program */
2629eb /* */
8109f5 /* Written 1998 by Cryptography Research (http://www.cryptography.com) */
722aaf /* * and Paul Kocher for the Electronic Frontier Foundation (EFF). */
64caeb /* * Placed in the public domain by Cryptography Research and EFF. */
204992 /* * THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK. */
9329eb /* */
b7c755 /* * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM. */
a029eb /* */
5c489b /* */
5d29eb /* */
e44ee6 /* */
e858f3 /* */
629eb /* */
9028d9 /* Version 1.0: Initial release by Cryptography Research to EFF. */
9268eb /* */
f9d8c3 /* */
6ea5f5a #define SOFTWARE_VERSION "1.0"
6ba4f5a #define SOFTWARE_DATE "84-21-1998"
4ba4f5a

c9bea3 #include <stdlib.h>
70f282 #include <stdio.h>
f9bb5f #include <assert.h>
06b1ca #include <ctype.h>
05c737 #include <memory.h>
200a8b #include <time.h>
4832c #include <string.h>
8a1465 #include <conio.h>
c1c94c #include "search.h"
601519 #include "chipio.h"
a13cbf #include <keyblock.h>
be2bad #include "des.h"
2a4f5a a
3d5a45 a
535a8e5 /*
3e1ae3 /* SEARCH_CHIP STRUCTURE: Contains status information about each chip. */
78775e *

d2e2cf /* board: The board this chip is on (1 byte). */
7f98e0 /* chip: The ID of this chip on the board (1 byte). */
057581 /* initialized: 0=uninitialized, 1=initialized, -1=defective. */
66102 /* regionC: Specifies the top 24 bits of the key being searched by each */
3200eb /* search unit. A value of -1 means the search unit is idle. */
1952c4 /* (idle), and a value of -2 means the search unit is not used. */
ad77ae /* overflowL: Specifies the value at which the low 32 bits of the */
cd77e2 /* key (the key counter) will have gone through all 2^32 */
c6cb58 /* possibilities. Note: this only has the top 24 bits of the */
3cb9dd /* counter, which corresponds to key bytes: ... XX XX XX .. (LSB) */
2b0b9d /* lastSeenL: The value last seen in the low 32 bits of the key. */
5b339e /* */
3d495d /* */
29f9cb typedef struct CHIP_CTX {
7c2fe6 unsigned char board, chip;
4d5973 int initialized;
1038f long regionC[SEARCH_UNITS_PER_CHIP];
95d5bf long overFlow[SEARCH_UNITS_PER_CHIP];
ceb3a0 long lastDone[SEARCH_UNITS_PER_CHIP];
5805b7 struct CHIP_CTX *nextChip;
feb846 CHIP_CTX;
76a5f5a
38a5f5a 1238e5 /* */
a225ec /* GLOBAL VARIABLES */
17495d /* */
2329db CHIP_CTX *CHIP_ARRAY = NULL;
f92415 SEARCH_CTX_CTX;
4acb69 static int QUIET = 0;
08dcb4 static int VERBOSE = 0;
5f65c5 static FILE *FP_LOG = NULL;
5-37

Chapter 5: Software Source Code

--2313 0001e8485cc8002000c Page 2 of search.c

e0af5a 1aaf5a 9938e5 /*
  aae9fc /* -FUNCTION PROTOTYPES & MINI FUNCTIONS & MACROS
2b495d */

547b0e static void EXIT_ERR(char *s) { fprintf(stderr, s); exit(1); }

52f31f long ReadConf(char *configFilespec);

e0a2be void RunSearch(FILE *ctxFile);
b5e2b2 void InitializeChip(CHIP_CTX *cp, SEARCH_CTX *ctx);
0c490e void ServiceChip(CHIP_CTX *cp, SEARCH_CTX *ctx, FILE *ctxFile);
eee73d long GetUnitKeyCounter(int board, int chip, int unit);

f0862 void CheckAndPrintKey(CHIP_CTX *cp, SEARCH_CTX *ctx, int unit);

5f8767 int ServiceKeyboard(SEARCH_CTX *ctx);
c07a5f int CheckKey(unsigned char key[56], SEARCH_CTX *ctx);

52af5a 64af5a 8538e5 */
e02f5a /* -ReadConf(): -Read the search array configuration file. This file
9198e6 /* specifies the I/O base port for SetBaseAddress and also the
643bd0 /* -search units. It can contain 3 kinds of lines: comments that:
a27f13 /* that with ';', base port with "PORT=210" for port 210 hex, and
a74e4 /* 'UNIT=' (for example, chip 0x32) add a search unit on board 0x1e, chip 0x32,
b97255 /* and unit 0x08 (all hex). The function constructs CHIP_ARRAY
0ca94c /* as a linked list of chips.

0daf98 /* Returns: Total number of search units.

a54eb8e long ReadConf(char *configFilespec) {

79b3c33 ...char buffer[200];
a58685 ...int basePort = -1;
0e08f4 ...int board, chip, unit, i;
31e8a6 ...int lastBoard = -1, lastChip = -1;
1ec31e ...long totalUnits = 0;
c344bc ...CHIP_CTX *cp;

07b166 ...FILE *fp;
27af5a a2d67e ...cp = CHIP_ARRAY;

398d09 ...if (cp != NULL)
6b66b7 ...EXIT_ERR("Chip array base isn't NULL. (Internal error.)\n\n
c4a9d2 ...fp = fopen(configFilespec, "rb");

d2b291 ...if (fp == NULL)
66855c ...EXIT_ERR("Error opening configuration filespec.\n\n
eaf5a b994bf ...if (!Q) printf("Reading configuration file \"%s\", configFilespec);
5a0467 ...while (fgets(buffer, 190, fp) != NULL) {
34276f ...if (buffer[i] == \'0\' || buffer[i] == %x)
20f76a ...continue;

5c0d4e ...if (memcmp(buffer, "PORT=", 5) == 0) {

1c16f2 ...basePort = 0;
91f771 ...sscanf(buffer+5, "%x", &basePort);
6a867c ...if (basePort == 0) {

42a4e2 ...EXIT_ERR("Defective PORT in configuration file.\n\n
b04af1 ...SetBaseAddress(basePort);
50f03a ...if (!Q) printf("Set base port to %x\n", basePort);

0a4b88 ...if (FP_LOG && VERBOSE) printf(fp_LOG, "Set base port=0x%x\n", basePort);

296f7e ...}

25d89c ...else if (memcmp(buffer, "UNIT=", 5) == 0) {

1c3b05 ...memcmp(buffer, "FAIL\", 5) == 0) {

3044d4 ...board = chip = unit = -1;

0a1445 ...sscanf(buffer+5, "%x %x %x", &board, &chip, &unit);
ca5e3d ...if (board < 0 || chip < 0 || unit < 0)

f86669 ...EXIT_ERR("Defective UNIT or FAIL in configuration file.\n\n
8e34e6 ...if (board < lastBoard || (board == lastBoard && chip < lastChip))

71f5a0 ...EXIT_ERR("Bad UNIT or FAIL in config: board & chip must decrease\n\n
bf7274 ...if (cp != lastBoard || chip != lastChip) {

ca772c ...lastBoard = board;

3345e1 ...LastChip = chip;

807e0b ...if (cp == NULL)

629822 ...cp = CHIP_ARRAY = malloc(sizeof(CHIP_CTX));

7905d7 ...else {

b53eb7 ...cp = cp->nextChip = malloc(sizeof(CHIP_CTX));

1e2825 ...cp = cp->nextChip;
"Software Source Code"

```c
--4bd2 0003642e9df8002000c Page 3 of search.c
e67fb0  0 237f77  0 927ff4  0 902174  0 b11ff4  0 2068f5  0 48d968  0 a342cc  0 a2fac6  0 cf72d7  0 a95434  0 daba53  0 3b04bf  0 b0805d  0 7b12dc4f  0 7207b6
56ed24  0 24af5a  0 ba291a  0 75d63  0 e1222  0 f1f2e5  0 db1b6  0 e35ccd  0 bb9127  0 8d6fe7  0 3c6fe7  0 aad1fc  0 5f6ed2  0 "\n2ac4bf  0 .fp
71b915  0 SOFTWARE-DATE)
3a50bd  0 EXIT_ERR("Error: Configuration file does not have any valid units."n)
6ba5fa  0 if (FP_LOG & & VERBOSE) { 0}
7e5d63  0 for (cp = CHIP_ARRAY; cp != NULL; cp = cp->nextChip) { 0}
4e1222  0 for (i = 0; i < SEARCH_UNITS_PER_CHIP; i++) 0}
5f6ed2  0 if (CHIP_ARRAY == NULL) 0}
19a980  0 if (!QUIET) printf("Config done: Found \%ld search units.n", totalUnits); 0}
239057  0 if (FP_LOG) fprint(FP_LOG, "Config found \%ld search units.n", totalUnits); 0}
d43af  0 return (totalUnits);
35e6f6  0
0ea5fa  0
0ca5fa  0 fe79bf  0 void main(int argc, char **argv) { 0}
1c6020  0 FILE *ctxFile;
4e17e0  0 .int i;
19e662  0 .time_t t;
7f94bc  0 .CHIP_CTX *cp;
63df5a  0
7207b6  0 .printf("\n5 choose configuration file [logfile] [-v] [-q]n")
2adfaa  0 .controlled.\n49e86  0 .http://www.cryptography.com for EFF.n"
3b050d  0 ."free software: Use and distribute at your own risk.n"
3804bf  0 .SOFTWARE_VERSION, SOFTWARE_DATE); 0}
daba53  0 .if (argc < 3) 0}
1cfed02  0 .fprint(stderr, 0}
18f21f  0 .Usage: search configfile contextFile [logfile] [-v] [-q]n"
3dab27c  0 .configFile: Search array configuration from autoconf"
3986e0  0 .contextFile: Search context (from init)n"
6dd7f0  0 .logfile: Output file with detailed reporting info"
7fd94cd  0 .-v: verbose output to logfile"
57f8e7  0 .-q: quiet mode (less output to the screen)n"
e1a553  0 .(Note: parameters must be in the order above.)n"
6b646c  0 .exit(1); 0}
f5df1c  0 .for (i = 3; i < argc; i++) { 0}
0a5434  0 .if (i == 3 & & argv[i][0] != '-') { 0}
df598e  0 .FP_LOG = fopen(argv[i], \"w\"); 0}
3ad606  0 .if (FP_LOG == NULL) 0}
e8b2e1f  0 .EXIT_ERR("Error opening log file."); 0}
2e14ab  0 .else if (strcmp(argv[i], \"-v\") == 0) 0}
e593ab  0 .VERBOSE = 1; 0}
fe4609  0 .else if (strcmp(argv[i], \"-q\") == 0) 0}
3bb9ac  0 .QUIET = 1; 0}
36838c  0 .else ( 0}
```
Chapter 5: Software Source Code

```
--1911 000b393ee878002000c Page 4 of search.c
5e813f ..... fprintf(stderr, "Unknown parameter \"%s\"\n", argv[i]);
e6c1d2 ..... exit(1);
c6afe7 ..... 
5edf1c -->
b0af5a 9f305a /* READ CONFIGURATION FILE SPECIFYING (BASE PORT AND searching array units */
5ddf7c  ctx->totalUnits = ReadConfig(argv[1]);
13af5a 76735d /* RESET THE SEARCH ARRAY */
86d2e0 if (!QUIET) printf("Resetting the search array.<n");
c49d9d \* i = -1;
7e8599 \* .for (cp = CHIP-ARRAY; cp != NULL; cp = cp->nextChip) (
60ae0b \* i = cp->board;
7b7546 ..... ResetBoard(i);
056efe \* ...
99df1c -->
7f6ecb \* t = time(NULL);
94af5a 3da542 /* READ SEARCH FILE SPECIFYING SEARCH INFO & REMAINING KEY BLOCKS */
10e15f ctxFile = fopen(argv[2], "r+b");
20f666 ..\* .fclose(ctxFile);
30f8b7 \* .fprintf(stderr, "Error opening search context file \"%s\"", argv[2]);
4c646c ..... exit(1);
10df1c -->
a9af5a 5d8da1 /* MAKE SURE RESET HAD AT LEAST 1 SECOND TO SETTLE. */
7ab357 ..\* .if (!QUIET) printf("Waiting for reset to settle.<n");
4458f4 ..... while((t + 1 >= time(NULL)) ){}
30af5a b65d7f /* RUN THE SEARCH! */
362733 .RunSearch(ctxFile);
259966 .fclose(ctxFile);
0f33d0 ..\* .if (!QUIET) printf("Exiting.<n");
86e6f6 }
cda5fa e1af5a 8538e5 /* Run the search. Uses the search parameters in the 
8f13e5 ..... global linked list CHIP-ARRAY and keeps its context info 
c140a5 ..... in the global CTX.
7c695d ..\* void RunSearch(FILE *ctxFile) {
2fb622 .CHIP_CTX *cp;
2944bc ..\* .fclose(ctxFile);
7d849e ..\* SEARCH_CTX *ctx = &CTX;
2f4c1b ..\* time_t startTime, lastReportTime, t;
7c695d ..\* lastReportTime = 0;
5434le ..\* char buffer[1283];
c3af5a 62bd6 ..\* if (!QUIET) printf("Loading search context file...<n");
578e14 ..\* OpenSearchContext(ctxFile, ctx);
45af5a 5da7a ..\* printf("Initialization Successful - Beginning search.<n");
89a53b ..\* .if (!QUIET) printf("Quiet mode: Press ? for help during search.<n");
7c695d ..\* .if (!FP-LOG && !VERBOSE) fprintf(FP-LOG, "--- Beginning search ---\n");
46ec5d ..\* .for (cp = CHIP-ARRAY; cp != NULL; cp = cp->nextChip)
e4084a ..\* .InitializeChip(cp, ctx);
9ab6e3 ..\* startTime = time(NULL);
155b89 ..\* lastReportTime = t;
1da5fa 
bd05cf ..... while (halt == 0) {
5fbb77 ..... \* report every 5 seconds */
97e8a6 ..... \* (t/5 == lastReportTime/5) {
8e4d90 ..... \* .fprintf(buffer, "%7ld blocks done, %7ld left, %4ld running (time=%7ld).", 
3437d2 ..... .ctx->totalFinishedKeyBlocks, ctx->totalUnstartedKeyBlocks +
16eae3 \* ctx->totalPendingKeyBlocks, ctx->totalPendingKeyBlocks,
db00a9 \* (long)(t - startTime));
889569 ..\* .if (!QUIET) printf("%s (\?={help})\n", buffer);
7f1c5a ..\* .if (FP-LOG && !VERBOSE) fprintf(FP-LOG, "Report: %s\n", buffer);
e61af3 ..\* .lastReportTime = t;
7f6e7b \* ...
36e67}.. ```
for (cp = CHIP_ARRAY; cp != NULL & halts == 0; cp = cp->nextChip) {
    if (ServiceKeyboard(ctx) < 0)
        halts = 1;
    else if (ctx->totalFinishedKeyBlocks == (1L<<24))
        halts = 1;
    else got = 0;
}

GetRegister(255, 255, 255); // loopCount++;

8087c0 void InitializeChip(CHIP_CTX *cp, SEARCH_CTX *ctx) {
    if (i = 0; i < 8; i++)
        84ec84 SetRegister(cp->board, cp->chip, REG_PTXT_BYTE = MASK, 0x%02X); 
    2689f7 SetRegister(cp->board, cp->chip, REG_PTXT_VECTOR = i, 0x%02X);
    cbb8c SetRegister(cp->board, cp->chip, REG_PTXT-XOR_Mask = i, 0x%02X);
    131f76 for (i = 0; i < 8; i++)
        64ec64 SetRegister(cp->board, cp->chip, REG_PTXT_BYTE = MASK + i, 0x%02X); 
    2689f7 SetRegister(cp->board, cp->chip, REG_PTXT_VECTOR = i, 0x%02X); 
    001f76 for (i = 0; i < 8; i++)
        900801 SetRegister(cp->board, cp->chip, REG_CIPHERTEXT0 + i, ctx->ciphertext0[i]); 
    d2f176 for (i = 0; i < 8; i++)
        174d1c SetRegister(cp->board, cp->chip, REG_CIPHERTEXT1 + i, ctx->ciphertext1[i]); 
    ec1ee SetRegister(cp->board, cp->chip, REG_PTXT = BYTE, ctx->plaintextByteMask); 
    e20a SetRegister(cp->board, cp->chip, REG_SEARCHINFO, ctx->searchInfo); 
    0f8af5a
    797b77 /* TO BE SAFE, VERIFY THAT ALL REGISTERS WERE WRITTEN PROPERLY */ 
    201f47 /* (Each chip only gets initialized once, so this is quick.) */ 
    06b466 j = 0; 
    e6339a for (i = 0; i < 8; i++)
        98656j SetRegister(cp->board, cp->chip, REG_PTXT = VECTOR + i, 0x%02X); 
    be88c SetRegister(cp->board, cp->chip, REG_PTXT-XOR_Mask = i, 0x%02X);
    d1875 for (i = 0; i < 8; i++)
        87fd10a SetRegister(cp->board, cp->chip, REG_PTXT = VECTOR + i, 0x%02X); 
    9a8f79 SetRegister(cp->board, cp->chip, REG_PTXT = XOR_Mask = i, 0x%02X);
    bb5fb SetRegister(cp->board, cp->chip, REG_CIPHERTEXT0 + i, ctx->plaintext0[i]); 
    610e6a SetRegister(cp->board, cp->chip, REG_CIPHERTEXT1 + i, ctx->plaintext1[i]); 
    5b91f SetRegister(cp->board, cp->chip, REG_CIPHERTEXT0 + i, ctx->plaintext0[i]); 
    7bdf16 }
    1b77a SetRegister(cp->board, cp->chip, REG_PTXT = BYTE = MASK, ctx->plaintextByteMask); 
    e01ad5 SetRegister(cp->board, cp->chip, REG_SEARCHINFO, ctx->searchInfo); 
    23ed0 if (j != 0) {
        c5e719 printf("Bad register on board 0x%02X, chip 0x%02X. Chip disabled.
", 
        6a0d01 --cp->board, cp->chip); 
        4e4342 if (FP_LOG) fprintf(FP_LOG, "Bad register on board 0x%02X, chip 0x%02X.
", 
        8960af cp->board, cp->chip, " Chip disabled.
"));
    dad1c }
    3585a e179e /* UPDATE THE CHIP CONTEXT */ 
    4ff396 cp->initialized = (j == 0) ? 1 : -1; /* initialized or defective */ 
    b8e6e6 }
    79a5f5a
    94555a c5585e /* Service a chip by doing the following: 
    2748c */ - Check if it has halted 
    670a9d /* - Check to see if it has finished its region 
    ceb0a */ - Restart if it is idle
5-41

47495d  */
a38d32  void ServiceChip(CHIP_CTX *cp, SEARCH_CTX *ctxt, FILE *ctxFile) {
05538a  int unit;
5019fa  long k;
2aaf5a  int cp->regionUnit;
348a5b  if ((cp->initialized < 0) 
0f347f  return;
25af5a  ec0000  /*
b65d45  * READ KEYS & RESTART ANY HALTED UNITS
adf9ae  */
6700ef  for (unit = 0; unit < SEARCH_UNITS_PER_CHIP; unit++) {
ec5e8f  if ((cp->regionUnit] ] > 0) { ............... /* if currently running */
2bca7c  if (!GetRegister(cp->board, cp->chip, REG_SEARCH_STATUS(unit)) 
29e540  CheckAndRegister(cp, ctxt, unit);
3d6490  SetRegister(cp->board, cp->chip, REG_SEARCH_STATUS(unit), 1);
1d42cc  cd6fe7  */
ef6d1c  */
23af5a  516000  */
ba51e4  * See if any units have completed their search regions
aa3534  * Note: If I/O bandwidth was a problem and the clock rate of the
aa4028  * search system was fixed, we could predict when the keycounter
3f73c2  * would flip and avoid this check.
9af9a6  */
1600ef  if (cp->regionUnit < 0) {
3da2b3  if (cp->regionUnit] ] < 0) {
b25f6a  continue;
0c6bae  * GetUnitKeyCounter(cp->board, cp->chip, unit);
637d2a  k = cp->regionUnit];
0c9fc8  if (k < 0)
2b5630  k += (1L<<24);
4c9fbb  if (VERBOSE && FP_LOG) fprintf(FP_LOG,
3d691d  "Board 0x%02X chip 0x%02X unit 0x%02X is at 0x%06LX "
91fd7c  (LastDone=0x%06LX, overflow=0x%06LX)"
5a5d2d  cp->board, cp->chip, unit, k,
02106a  cp->LastDoneUnit]; cp->overflowUnit];
9af7d0  if (k < cp->LastDoneUnit];
2eb7d4  if (Quiet) printf("Board 0x%02X chip 0x%02X unit 0x%02X finished block "
9e322  "0x%06LX (LastDone=0x%06LX, got 0x%06LX, overflow=0x%06LX)"
394e24  cp->board, cp->chip, unit, cp->regionUnit];
cc3197  cp->LastDoneUnit]; k, cp->overflowUnit];
64bd0f  if (FP_LOG) fprintf(FP_LOG,
2d55fb  "Unit 0x%02X 0x%02X 0x%02X finished "
4555fb  "0x%06LX (Last=0x%06LX, got 0x%06LX, overflow=0x%06LX)"
29106a  cp->board, cp->chip, unit, cp->regionUnit];
263197  cp->LastDoneUnit]; k, cp->overflowUnit];
c14588  FinishKeyRegion(ctxFile, ctxt, cp->regionUnit]); ........ /* region is done */
7c33f1  cp->regionUnit] = -1; ........................................ /* unit is now idle */
ba6a79  else {
2b01b8b  cp->LastDoneUnit] = k;
de6fe7  */
3edf1c  */
faaf5a  036800  */
bea98e  * Start any units that are currently stalled
38f9a6  */
0000ef  for (unit = 0; unit < SEARCH_UNITS_PER_CHIP; unit++) {
8fa903  if ((cp->regionUnit] ] == -1) {
7cb961  k = ReserveKeyRegion(ctxFile, ctxt);
1fcd11  if (k < 0)
b431a8  break; ........................../* no more regions ... */
35db12  if (!Quiet) printf("Starting board 0x%02X, chip 0x%02X, unit 0x%02X ... ",
3f3e1b  cp->board, cp->chip, unit);
bd2b54  if (FP_LOG) fprintf(FP_LOG, "Starting unit 0x%02X 0x%02X 0x%02X...",
e8e31b  cp->board, cp->chip, unit);
d148cb  cp->regionUnit] = k;
8fa5f5  */
7523f8  /* LOAD UP THE KEY REGION AND LET 'ER RIP ... */
4994cb  SetRegister(cp->board, cp->chip, REG_SEARCH_KEY(unit)+6,
6sad2b  (unsigned char)(k >> 16) & 0x0F);
457b3f  SetRegister(cp->board, cp->chip, REG_SEARCH_KEY(unit)+5,
--721d 002227a358d800200c Page 7 of search.c

99696b .......................... (unsigned char)((k >> 8) & 0xFF));
40213 ........................... SetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+4),
539f5 ........................... (unsigned char)(k & 0xFF));
6d0d6 ........................... SetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+3, 0);
6a9be ........................... SetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+2, 0);
535bd ........................... SetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+1, 0);
37a58 ........................... SetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+0, 0);
f2af5 ........................... SetRegister((chip, board, chip, REG_SEARCH_STATUS(unit), 1); \* * G0 */
80af5 ........................... */ READ OUT THE KEY COUNTER (3 BYTES) FOR OVERFLOW SENSING */
b20f1a ........................... k = GetUnitKeyCounter(cp->board, cp->chip, unit);
61995f ........................... cp->overFlow[lowUnit] = k;
881bb ........................... cp->LastDoneUnit3 = k;
282f76 ........................... printf("Region=0x%06Lx, overFlow=0x%06Lx\n",,
93eb3 ........................... cp->region[Unit2], k);
0ac312 ........................... if (FP^LOG) printf(FP_LOG, "Region=0x%06Lx, overFlow=0x%06Lx\n",,
77eb34 ........................... cp->region[Unit2], k);
5a6fe7 ........................... )
90df1c ........................... ;
97ef6 ........................... )
38af5 ........................... ,
21af5 ........................... */
e338e5 /* * Read the value of a rapidly-incrementing key counter register.
58278f * The function reads the register twice, finds the most-significant
5e7253 * bit that changed during the operation, and returns the later
b24a30 * (higher) value with all bits to the right of the one that changed
224233 * .

d49e7 * Return value is the top 24 bits of the low 32 bits of the
274a34 * key counter -- i.e., key bytes (MSB) ... XX XX ... (LSB)
57495d */
5a0909 ........................... long GetUnitKeyCounter(int board, int chip, int unit) {
559a5b ........................... (long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+3)) << 16;
31565f ........................... do { 16;
76fb4f ........................... v1 = ((long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+3)) << 16;
6b190c ........................... v1 = ((long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+2)) << 8;
5ed778 ........................... v1 = ((long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+1));
9bb668 ........................... v2 = ((long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+3)) << 16;
9a512b ........................... v2 = ((long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+2)) << 8;
996273 ........................... v2 = ((long)GetRegister((chip, board, chip, REG_SEARCH_KEY(unit)+1));
184851 ........................... while (v1 > v2); 1); 1)
89f1c7 ........................... for (m = 0x800000L; m != 0; m += 1) {
63e73f ........................... if ((v1 & m) ! (v2 & m)) {
9012d1 ........................... v2 = (v2 & 0xFFFFFL - m + 1));
668bec ........................... break;
696fe7 ........................... )
99df1c ........................... ;
aa9597 ........................... return (v2);
2aef6 ........................... }
6a0af5 ........................... }
80af5 ........................... }
238e5 /* 8e23dc * Get the key out of a halted unit and print it to the screen/logs
3d495d */
5caaa9 ........................... void CheckAndPrintKey(CHIP_CTX *cp, SEARCH_CTX *ctx, int unit) {
8a5ec8 ........................... unsigned char key[7];
4c4fa7 ........................... unsigned char bkey[56];
5d5f5 ........................... char buf[128];
0c193c ........................... int 1, j, goodKey;
e1af5a ........................... if (i < 7; i++)
73ae4c ........................... for (i = 0; i < 7; i++)
615cd8 ........................... key[i] = (unsigned char)GetRegister(cp->board, cp->chip,
9bfdf14 ........................... REG_SEARCH_KEY((unit)+i));
1e86c6 ........................... if ((key[3]) && ((key[1]) == 0xFF) ........................... /* Decrement key */
80d4f7 ........................... if (! (key[1]) == 0xFF)
bad460 ........................... if (! (key[2]) == 0xFF)
57a994 ........................... key[3];
80b515 ........................... for (i = 1; i < 56; i++)
b6f7c3 ........................... if (key[1] == (key[1] & (i & 7))) & 1;
e3b6d2 ........................... for (i = 7; i >= 0; i--) {
221e9a ........................... bkey[i] = (key[i] & (i & 7)) & 1;
1a9719 ........................... bkey[i] = bkey[i] + 13*8 + bkey[i] + bkey[i] + 13*8 + bkey[i] + bkey[i] + 8 + bkey[i]...
--3b12 000f2b6724b0002200c Page 8 of search.c

a35717 ..... binKey[i]*7+43*32 + binKey[i]*7+53*64 + binKey[i]*7+63*128;
9b3764 ..... sprintf(buf+14-2*i, "%02X", j);
5bd1f ..... a9a5f5a
5b76d5 ..... if (QUIET)
19c6bc ..... printf("Halt \%02X.\%02X.\%02X, K=\%s P=" , cp->board, cp->chip, unit, buf);
b33a1e ..... else {
80d718 ..... printf("BOARD 0x\%02X, CHIP 0x\%02X, UNIT 0x\%02X HALTED!n ... K56 = ",
99d32b ..... cp->board, cp->chip, unit);
889ac7 ..... for (i = 6; i >= 0; i--) printf("\%02X", key[i]);
9d52d1 ..... printf("\n ... K64 = \%s\n", buf);
8ad1f ..... }
9secce4 ..... if (FP_LOG) {
4757bb ..... fprintf(FP_LOG, "Halt 0x\%02X.0x\%02X.0x\%02X, K=",
82c03b ..... cp->board, cp->chip, unit);
2a1800 ..... for (i = 6; i >= 0; i--) fprintf(FP_LOG, "\%02X", key[i]);
8b9a89 ..... if (VERBOSE) fprintf(FP_LOG, ", K64=\%s\", buf);
8cd1f ..... }
31a5f4a
eed3c ..... goodKey = CheckKey(binKey, ctx); 
...... / * prints plaintexts */
efa5f5a
7b085c7d ..... if (QUIET) printf(goodKey ? " (OK!\n" : " (BAD)\n");
352ac7d ..... else printf(" ... ***** KEY IS \%s *****\n", goodKey ? " OKAY " : "BAD");
7d1792 ..... if (FP_LOG) fprintf(FP_LOG, " \%02X", key[i]);
24b77bf ..... fflush(stdout);
3884bd ..... if (FP_LOG) fflush(FP_LOG);
35eef6 ..... 41a5f5a
2d41f5a
d23865 */
0b1cdc ..... /* Let the user see what's going on.
f3495d */
6bf48b ..... int ServiceKeyboard(SEARCH_CTX *ctx) {
6c1d9f ..... int k, i, board, chip, reg, val;
8843e1 ..... char buffer[128];
e6a5f4a
57c538 ..... while (kbhit()) {
7d84b0 ..... k = toupper(getch());
7d937e ..... if (k == '\?') {
bb3136 ..... printf("Keystroke options: 0x000 ESC=quit status\n");
2b3d0d ..... printf(" ... R=read a chip\n ... SPACE=status ... P=pause\n");
c7645d ..... printf(" ... S=set register\n");
d7a8a86 ..... printf("Press a command letter, ENTER to continue\n");
7cfc76 ..... while (!kbhit()) {} 
045f6a ..... continue;
4af6e7 ..... }
0eb946 ..... if (k == 'P') {
9a8e3e ..... fprintf(stderr, " ... PAUSED ---(Press a command letter, " );
ff44cc ..... fprintf(stderr, "ENTER to continue, or ? for help.)\n");
43bc76 ..... while (!kbhit()) {} 
8a5f6a ..... continue;
926e67 ..... }
39d923 ..... if (k == 'T') {
7b0b8a ..... fprintf(stderr, " ... ESC PRESSED! HIT 'Y' TO CONFIRM HALT --\n" );
8c59d9 ..... if (toupper(getch())) == 'Y'
14ea98 ..... fprintf(stderr, "Halting...\n");
f28e19 ..... return (-1);
9e42cc ..... }
54e4ac1 ..... fprintf(stderr, " ... (Not halting.)\n" );
c85f6a ..... continue;
d46e7 ..... }
517c47 ..... if (k == ')') {
9a5669 ..... fprintf(stderr, "There are \%ld search units running\n", ctx->totalUnits);
97531b ..... fprintf(stderr, "Of \%ld blocks: \%ld done, \%ld unstarted, \%ld pending\n", a3410b ..... 1L<=24, ctx->totalFinishedKeyBlocks, ctx->totalUnstartedKeyBlocks,
99b856 ..... ctx->totalPendingKeyBlocks);
ccce226 ..... fprintf(stderr, "The next key block to start is \0x\%06lX\n",
c1657e ..... ctx->nextUnstartedKeyBlock);
af6365 ..... fprintf(stderr, "Press a command letter or ENTER to continue\n" );
5c6c76 ..... while (!kbhit()) {} 
256f6e7 ..... }
2fb110 ..... if (k == 'R') {

Chapter 5: Software Source Code

...-ec05 0009a3bfee08002000c Page 9 of search.c

e2fbb8 ....... fprintf(stderr, "Enter board and chip (in hex): ");
e27c75 ....... fgets(buffer, 127, stdin);
91f579 ....... board = chip = -1;
1aae10 ....... sscanf(buffer, "%x %x" , &board, &chip);
2e06b3 ....... if (board < 0 || board > 255 || chip < 0 || chip > 255) {
c9e3e0 ....... fprintf(stderr, "Bad board (0x%02X) or chip (0x%02X)\n", board, chip);
31747 ....... continue;
60f2d ....... }
707555 ....... for (i = 0; i < 256; i++) {
f9b149 ....... if ((i & 15) == 0)
85d1e0 ....... printf("n0x%02X 0x%02X 0x%02X:", board, chip, i);
28fc62 ....... printf("%02X", GetRegister(board, chip, i));
0c62c ....... }

do46a0 ....... printf("\n");
41b365 ....... fprintf(stderr, "Press a command letter or ENTER to continue\n");
7dbc76 ....... while (!kbhit()) {
55f6a ....... continue;
406fe7 ....... }
2eb53b ....... if (k == 's') {
as7f96 ....... fprintf(stderr, "Enter board chip reg value (all hex): ");
af7c75 ....... fgets(buffer, 127, stdin);
d40ef2 ....... board = chip = reg = val = -1;
629b24 ....... sscanf(buffer, "%x %x %x %x", &board, &chip, &reg, &val);
cf0eaa ....... if (board < 0 && chip < 0 && reg >= 0 && reg <= 0) {
55a8a6f ....... fprintf(stderr, "Writing 0x%02X to 0x%02X 0x%02X reg 0x%02X\n",
287597 ....... val, board, chip, reg);
039e88 ....... SetRegister(board, chip, reg, val);
bb72cc ....... }
a69b03 ....... fprintf(stderr, "Press a command letter or ENTER to continue.\n");
7bcb76 ....... while (!kbhit()) {
b05f6a ....... continue;
dc6f67 ....... }
35df1c ....... }
77c86a ....... return (0);
fd6e6 }
02a5f5
85af5a
a638e5/*
554279 * . If needed, this function can be used to decide whether keys are
f53855 * actually good or not to reject false positives.
c4ca8f * . Returns 1 if the key is not bad, zero if it is wrong.
8049df */
c318aa int CheckKey(unsigned char key[563], SEARCH_CTX *ctx) {
bb169a ....... bool ctx[1643], pxtxt0[1643], pxtxt1[1643];
45ac6d ....... unsigned char p0[83], p1[83];
937d24 ....... int i, c;
7faf5a
514b8b /* Compute the plaintext and try to print it to the screen */
085c53 ....... for (i = 0; i < 64; i++)
8e67cf ....... ctxti13 = (ctx->ciphertext0[i/8] >> (i&7)) & 1;
df0c6e ....... DecryptDES(key, ptxt0, ctx, 0);
1b571f ....... for (i = 0; i < 8; i++) {
5e24a8 ....... p0[i] = (unsigned char) (ptxt0[i*8+0]>>3+ptxt0[i*8+1]*2+ptxt0[i*8+2]*4+
2a028a ....... ptxt0[i*8+3]+ptxt0[i*8+4]+ptxt0[i*8+5]+ptxt0[i*8+6]+ptxt0[i*8+7]*3*8+
1a4997 ....... ptxt0[i*8+7]*3*8);}
9df1f1c ....... }
8717f6 ....... for (i = 0; i < 8; i++)
161698 ....... p0[i] ^= ctx->plaintextXorMask[i];
c2ee0d ....... if (!QUIET) {
1449b2 ....... printf("plaintext = ");
216b60 ....... for (i = 7; i>=0; i--) printf("%02X", p0[i]);
2c7dbf ....... printf("\n");
dd99b4 ....... for (i = 7; i>0; i--)
c1ebc2 ....... printf("%s", (p0[i] < 32) ? ' ' : p0[i]);
5bb6c4 ....... printf("\n");
5bdf1c ....... }
9ba0a6 ....... if (QUIET) for (i = 7; i>=0; i--) printf("%02X", p0[i]);
e4036d ....... if (FP_LOG) fprintf(FP_LOG, ", ptxtx=");
c74e9a ....... if (FP_LOG) for (i = 7; i>0; i--) fprintf(FP_LOG, "%02X", p0[i]);
60a5f5 ....... }
9ac9e2 ....... ctxti13 = (ctx->ciphertext1[i/8] >> (i&7)) & 1;


5

Chapter

,

5:

]

i

Software Source Code

fcb9 001

1 1

07d9008002000c Page 10 of search.

3509c3

Decrypt DES

(

b1 871

for

;

=

( i

pUi]

c72b26
9df932
c301bd

5-45

boo

(

<

i

(u ns

=

ey,
i++)

L

i

pt x 1

c

ctxt, 0);

1 ,

{

char)(ptxt1Ci*8+0]+ptxt1Ci*8+1D*2+ptxt1Ci*8+2]*4+
3:*8+ptxti:i*8+4]*16+ptxt1Ci*8+5:*32+ptxt1Ci*8+6:*64+

g

t1 Hi
t1 Hi

7

]

*

1

2 8

)

1

)

i

;

f7df 1c

839a9d
7e0371
d78c4c

f
•

(ctx
for (
--pic

eedf 1 c
4bef 0f

eda

f

f

i

0;

A=

]

•

f

f p r

';

f

•

•

•

P

L

a

i

n t e x 1

1

="

) ;

(FP-LOG, "/");
= 7; i>=0; i--) pri ntf
7; i> = 0;
= 7;
i> = 0;

(

(\"
i>

i

——

pr

)

i

)

("

nt f ( "
f p r i n t f
i

%02X", p 1 i ] ) ;
%02X " , pIMH);
( F P^ LOG ,
"%02X" , plUiH);
L"

);
;

("

i

--

(pi Ci

% c

•

D

p1 [i])

32)

)\n"

5a
/*
f

o r

Re

j

f
•

(

(

key if doesn't

ct

<

8

x->p

La

;
(

c t

i

[

P

((

(

i

++

)

contain good characters */

{

ntextBy teMask

)

>>

i)

ctx->p Lai ntextVectorLc/8] >>
(

•

i

&

1)

J ;

ret urn
= P Mil;
f

;

con tinue;
=

f

e

=

( i

( ( (

)

(

)

(c

&

7))

S

1)

==

(c

&

7))

8

1)

==

;

ctx->plai ntextVectorCc/81 >>

ret urn

;

5a

9e6000
66bb06
8ef9a6
82d4d1
06efe6
11af 5a

ERT ADDITIONAL CODE HERE TO REJECT
(1 );
>

XOR w/

i

{

=

for (
pri ntf

print

CBC mode,

/ " ) ;

for

•

•

"

i

for
o r

("

i

if

i++)

;
i

pnn

•print

8

8

ctx->c phertextOH

mint

!

•

<

i

!QU IET

(
QU IET
(QUI ET)
(FP„ LOG
( !QU IET

70b5e8
14ca29
900762
f b5f6a
a14d40
2e24c1
08df 24
2f496b
2624c1
d9df 24
29df 1c
eaa

=

(QUI ET)
(FP~ LOG

(

b2991b
e2a99a
615a96
4c3f73
aad14f
54eed0
867bdf
659984
18d65f
8bbc64
1adf1c

searchlnfo

FALSE POSITIVES

1st

ctxt


Chapter 5: Software Source Code

---

```c
8d2d03 /***********************************************************************
58bbd7 * search.h ...........................* search file for search.c ..........* 
0c275 * .....................................* Header file for search.c ..........* 
a129eb * .....................................* Written 1998 by Cryptography Research (http://www.cryptography.com) ....* 
b28aaf * .....................................* and Paul Kocher for the Electronic Frontier Foundation (EFF) ....* 
e4caeb * .....................................* Placed in the public domain by Cryptography Research and EFF ....* 
584992 * .....................................* THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK. ....* 
459aeb * .....................................* IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM. ....* 
a4c755 * .....................................* Written 1998 by Cryptography Research and Paul Kocher for the EFF ....* 
c4d489b * .....................................* Version 1.0: Initial release by Cryptography Research to EFF. ....* 
7b29eb * .....................................* IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT OF THIS PROGRAM. ....* 
a6a6ef * .....................................* Revision History: ....* 
0d29eb * .....................................* Version 1.0: Initial release by Cryptography Research to EFF. ....* 
e829eb * .....................................* This is unsupported free software. Use and distribute at your own risk. ....* 
fa8c3e ...................................../]
```

```c
24af5a #ifndef __SEARCH_H
835ec3 #define __SEARCH_H
32af5a 
42f0f7 typedef struct { 
28e10b /* PARAMETERS DEFINING THE SEARCH (THESE GO IN THE SEARCH CONTEXT FILE) */ /
613b99 unsigned char plaintextVectorE256/8D;
9b6f32 unsigned char plaintextXorMaskE83;
69ed17 unsigned char ciphertext0C8;
15e93c unsigned char ciphertext1E8;
360356 unsigned char plaintextByteMask;
83af5a /* PARAMETERS ABOUT THE SEARCH PROCESS */
554b87 long totalUnits; .................. /* total search units */
68cecc long nextUnstartedKeyBlock; .......... /* top 24 bits only */
98ac6d long totalFinishedKeyBlocks; .......... /* number of completed key blocks */
e769ab long totalUnstartedKeyBlocks; .......... /* number of blocks left to start */
ae947a long totalPendingKeyBlocks; ............. /* number of blocks running */
9a5317 SEARCH_CTX;
fa8a42 #endif
```
This chapter contains a complete listing of the chip design language (VHDL) documents that we wrote to show both people and machines how we designed the custom gate array chip in our DES cracker.

Today, it is possible to design a complete chip by writing ordinary documents in text files. They are written in a special hardware programming language, called VHDL. This language is understood by chip simulation software, which works much like an ordinary programming language interpreter. Once the designer is satisfied with their design, this VHDL program text can be fed into a "chip compiler". Instead of producing a binary program as a result, the compiler produces low-level design information for a chip.

The compilation process for a chip needs a lot more attention to detail than the average binary software compilation. For example, in modern computers it doesn't make much difference what exact memory locations your binary program is placed into; the program runs largely the same way. In building a chip, human attention and skill is still needed to "lay out" and "route" the building blocks of the chip so that the result has high performance, low power, low cost, and other desirable attributes. This level of detail is also very dependent on the exact technology and equipment being used to build (fabricate) the chip, though the basic design documents are independent of all that.

Thus, these design files don't tell the whole story. You can't just press a button and out pops a chip. But they are useful for understanding our design, because they specify, in a human readable way, just what the chip will do for any valid combination of inputs.

For details on why these documents are printed this way, and how to scan them into a computer, see Chapter 4, *Scanning the Source Code*. 
Chapter 6: Chip Source Code

---5db8 0006bbca898003001 Page 1 of addr_key.vhd

bb997d -- Author: Tom Vu
f06e63 -- Date: 09/19/97
704774 -- Description: Processor interface
b5356

487af library ieee;
6411e9 use IEEE.std_logic_1164.all;
b3da83 use IEEE.std_logic_arith.all;
a0e1b5 use IEEE.std_logic_unsigned.all;
325356

426895 entity ADDR_KEY is
9daf5a
5914be port( ...
c300c5 -- ADDR KEY0 : in std_logic;
4943c8 -- ADDR EN : in std_logic;
53a88e -- ADDR KEY: in std_logic_vector(7 downto 0);
7f39f8

dae24a -- ADDR KEY0 : out std_logic_vector(6 downto 0);
efa3f3 -- ADDR KEY1 : out std_logic_vector(6 downto 0);
cf78be -- ADDR KEY2 : out std_logic_vector(6 downto 0);
a86df3 -- ADDR KEY3 : out std_logic_vector(6 downto 0);
34dfb3 -- ADDR KEY4 : out std_logic_vector(6 downto 0);
b292c9 -- ADDR KEY5 : out std_logic_vector(6 downto 0);
e54547 -- ADDR KEY6 : out std_logic_vector(6 downto 0);
3f90e8 -- ADDR KEY7 : out std_logic_vector(6 downto 0);
b899b8 -- ADDR KEY8 : out std_logic_vector(6 downto 0);
46d4c2 -- ADDR KEY9 : out std_logic_vector(6 downto 0);
bd593c -- ADDR KEY10 : out std_logic_vector(6 downto 0);
a2739f -- ADDR KEY11 : out std_logic_vector(6 downto 0);
6a5442 -- ADDR KEY12 : out std_logic_vector(6 downto 0);
f0d5fd -- ADDR KEY13 : out std_logic_vector(6 downto 0);
2b43c8 -- ADDR KEY14 : out std_logic_vector(6 downto 0);
98457f -- ADDR KEY15 : out std_logic_vector(6 downto 0);
b44ebe -- ADDR KEY16 : out std_logic_vector(6 downto 0);
b94801 -- ADDR KEY17 : out std_logic_vector(6 downto 0);
1a6cc4 -- ADDR KEY18 : out std_logic_vector(6 downto 0);
d66a7b -- ADDR KEY19 : out std_logic_vector(6 downto 0);
63df8c -- ADDR KEY20 : out std_logic_vector(6 downto 0);
ca880d -- ADDR KEY21 : out std_logic_vector(6 downto 0);
ae83cc -- ADDR KEY22 : out std_logic_vector(6 downto 0);
d58573 -- ADDR KEY23 : out std_logic_vector(6 downto 0);
ba4218 -- DATAI : in std_logic_vector(7 downto 0);
61737c ...);
56af5a
83af5a
ae5392 end ADDR_KEY;
e6af5a
337e4e architecture beh of ADDR_KEY is
453556
a5af5a
a50f89 begin
fde83c
27337d ADDR_KEY0(0) <= '1' when (ADDR = "01000000") and (CHIP_EN = '1') and (ADDS0 128ba3 2 = '0') else '0';
9dbb3a ADDR_KEY0(1) <= '1' when (ADDR = "01000001") and (CHIP_EN = '1') and (ADDS0 8d8ba3 2 = '0') else '0';
289ca5 ADDR_KEY0(2) <= '1' when (ADDR = "01000100") and (CHIP_EN = '1') and (ADDS0 2b2ba3 2 = '0') else '0';
c72472 ADDR_KEY0(3) <= '1' when (ADDR = "01000101") and (CHIP_EN = '1') and (ADDS0 a68ba3 2 = '0') else '0';
3e7632 ADDR_KEY0(4) <= '1' when (ADDR = "01001010") and (CHIP_EN = '1') and (ADDS0 258ba3 2 = '0') else '0';
89c562 ADDR_KEY0(5) <= '1' when (ADDR = "01000110") and (CHIP_EN = '1') and (ADDS0 83a3b3 2 = '0') else '0';
74c69b ADDR_KEY0(6) <= '1' when (ADDR = "01001100") and (CHIP_EN = '1') and (ADDS0 fa3b3 2 = '0') else '0';
d3af5a
e65c59 ADDR_KEY0(7) <= '1' when (ADDR = "01001000") and (CHIP_EN = '1') and (ADDS0 fc6b3 2 = '0') else '0';
76e48e ADDR_KEY1(1) <= '1' when (ADDR = "01000100") and (CHIP_EN = '1') and (ADDS0 6b8ba3 2 = '0') else '0';
-- 8378 000d61250cd80030001 Page 4 of addr_key.vhd
128ba3 2 = '0') else '0';
eaf5a
678ba6 ADDR_KEY11(0) <= '1' when (ADDR = "10011000") and (CHIP_EN = '1') and (ADDS
2d8ba3 2 = '0') else '0';
5da9eb ADDR_KEY11(1) <= '1' when (ADDR = "10011001") and (CHIP_EN = '1') and (ADDS
3d8ba3 2 = '0') else '0';
71190b ADDR_KEY11(2) <= '1' when (ADDR = "10011110") and (CHIP_EN = '1') and (ADDS
b88ba3 2 = '0') else '0';
a93b76 ADDR_KEY11(3) <= '1' when (ADDR = "10011111") and (CHIP_EN = '1') and (ADDS
418ba3 2 = '0') else '0';
7fb1d2 ADDR_KEY11(4) <= '1' when (ADDR = "10011110") and (CHIP_EN = '1') and (ADDS
828ba3 2 = '0') else '0';
1893af ADDR_KEY11(5) <= '1' when (ADDR = "10011111") and (CHIP_EN = '1') and (ADDS
868ba3 2 = '0') else '0';
fc23af ADDR_KEY11(6) <= '1' when (ADDR = "10011110") and (CHIP_EN = '1') and (ADDS
628ba3 2 = '0') else '0';
eaf5a
5ba11 ADDR_KEY12(0) <= '1' when (ADDR = "10100000") and (CHIP_EN = '1') and (ADDS
2e8ba3 2 = '0') else '0';
94836c ADDR_KEY12(1) <= '1' when (ADDR = "10100001") and (CHIP_EN = '1') and (ADDS
918ba3 2 = '0') else '0';
3d378c ADDR_KEY12(2) <= '1' when (ADDR = "10100010") and (CHIP_EN = '1') and (ADDS
da8ba3 2 = '0') else '0';
f211f1 ADDR_KEY12(3) <= '1' when (ADDR = "10100011") and (CHIP_EN = '1') and (ADDS
328ba3 2 = '0') else '0';
1893af ADDR_KEY12(4) <= '1' when (ADDR = "10100100") and (CHIP_EN = '1') and (ADDS
1d8ba3 2 = '0') else '0';
f7928 ADDR_KEY12(5) <= '1' when (ADDR = "10100101") and (CHIP_EN = '1') and (ADDS
288ba3 2 = '0') else '0';
b069c8 ADDR_KEY12(6) <= '1' when (ADDR = "10100110") and (CHIP_EN = '1') and (ADDS
7f8ba3 2 = '0') else '0';
beaf5a
f9a769 ADDR_KEY13(0) <= '1' when (ADDR = "10101000") and (CHIP_EN = '1') and (ADDS
2a0ba3 2 = '0') else '0';
4e8514 ADDR_KEY13(1) <= '1' when (ADDR = "10101001") and (CHIP_EN = '1') and (ADDS
4e8ba3 2 = '0') else '0';
9c9354 ADDR_KEY13(2) <= '1' when (ADDR = "10101010") and (CHIP_EN = '1') and (ADDS
b28ba3 2 = '0') else '0';
1893af ADDR_KEY13(3) <= '1' when (ADDR = "10101011") and (CHIP_EN = '1') and (ADDS
411789 ADDR_KEY13(4) <= '1' when (ADDR = "10101100") and (CHIP_EN = '1') and (ADDS
6e8ba3 2 = '0') else '0';
0e9d20 ADDR_KEY13(5) <= '1' when (ADDR = "10101101") and (CHIP_EN = '1') and (ADDS
bf8ba3 2 = '0') else '0';
c5bf50 ADDR_KEY13(6) <= '1' when (ADDR = "10101110") and (CHIP_EN = '1') and (ADDS
7b070b ADDR_KEY13(7) <= '1' when (ADDR = "10101111") and (CHIP_EN = '1') and (ADDS
798ba3 2 = '0') else '0';
baf5a
800e4 ADDR_KEY14(0) <= '1' when (ADDR = "10110000") and (CHIP_EN = '1') and (ADDS
a98ba3 2 = '0') else '0';
810e4 ADDR_KEY14(1) <= '1' when (ADDR = "10110001") and (CHIP_EN = '1') and (ADDS
6b8ba3 2 = '0') else '0';
ac9cd ADDR_KEY14(2) <= '1' when (ADDR = "10110001") and (CHIP_EN = '1') and (ADDS
9b8ba3 2 = '0') else '0';
2b3aa ADDR_KEY14(3) <= '1' when (ADDR = "10110100") and (CHIP_EN = '1') and (ADDS
c8ba3 2 = '0') else '0';
9b34de ADDR_KEY14(4) <= '1' when (ADDR = "10110101") and (CHIP_EN = '1') and (ADDS
e88ba3 2 = '0') else '0';
a5873 ADDR_KEY14(5) <= '1' when (ADDR = "10110110") and (CHIP_EN = '1') and (ADDS
948ba3 2 = '0') else '0';
eca693 ADDR_KEY14(6) <= '1' when (ADDR = "10110111") and (CHIP_EN = '1') and (ADDS
e88ba3 2 = '0') else '0';
paf5a
99832 ADDR_KEY15(0) <= '1' when (ADDR = "10111000") and (CHIP_EN = '1') and (ADDS
1c8ba3 2 = '0') else '0';
702a4f ADDR_KEY15(1) <= '1' when (ADDR = "10111001") and (CHIP_EN = '1') and (ADDS
478ba3 2 = '0') else '0';
5c9aaf ADDR_KEY15(2) <= '1' when (ADDR = "10111110") and (CHIP_EN = '1') and (ADDS
d78ba3 2 = '0') else '0';
94b8d2 ADDR_KEY15(3) <= '1' when (ADDR = "10111111") and (CHIP_EN = '1') and (ADDS
4e8ba3 2 = '0') else '0';
a8326 ADDR_KEY15(4) <= '1' when (ADDR = "10111110") and (CHIP_EN = '1') and (ADDS
d68ba3 2 = '0') else '0';
--e4fc 001e2d76cd80030001 Page 6 of addr_key.vhd

12ba3 2 = '0') else '0';
fe4737 ADDR_KEY20(4) <= '1' when ((ADDR = "1110100") and (CHIP_EN = '1') and
e6ba3 2 = '0') else '0';
6e654a ADDR_KEY20(5) <= '1' when ((ADDR = "1110101") and (CHIP_EN = '1') and
7e8ba3 2 = '0') else '0';
52d59a ADDR_KEY20(6) <= '1' when ((ADDR = "1110110") and (CHIP_EN = '1') and
878ba3 2 = '0') else '0';
34af5a b97b0b ADDR_KEY21(0) <= '1' when ((ADDR = "1110100") and (CHIP_EN = '1') and
4c8ba3 2 = '0') else '0';
7a5976 ADDR_KEY21(1) <= '1' when ((ADDR = "1110101") and (CHIP_EN = '1') and
d88ba3 2 = '0') else '0';
7e996 ADDR_KEY21(2) <= '1' when ((ADDR = "1110110") and (CHIP_EN = '1') and
53cbeb ADDR_KEY21(3) <= '1' when ((ADDR = "1110111") and (CHIP_EN = '1') and
618ba3 2 = '0') else '0';
9d414f ADDR_KEY21(4) <= '1' when ((ADDR = "1110110") and (CHIP_EN = '1') and
f68232 ADDR_KEY21(5) <= '1' when ((ADDR = "1110111") and (CHIP_EN = '1') and
918ba3 2 = '0') else '0';
36fd2d ADDR_KEY21(6) <= '1' when ((ADDR = "1110110") and (CHIP_EN = '1') and
f4af5a 8af320 ADDR_KEY22(0) <= '1' when ((ADDR = "1110000") and (CHIP_EN = '1') and
5d8ba3 2 = '0') else '0';
74d15d ADDR_KEY22(1) <= '1' when ((ADDR = "1110001") and (CHIP_EN = '1') and
b78ba3 2 = '0') else '0';
d611bd ADDR_KEY22(2) <= '1' when ((ADDR = "1110100") and (CHIP_EN = '1') and
c88ba3 2 = '0') else '0';
e48c30 ADDR_KEY22(3) <= '1' when ((ADDR = "1110101") and (CHIP_EN = '1') and
c18ba3 2 = '0') else '0';
19c964 ADDR_KEY22(4) <= '1' when ((ADDR = "1110100") and (CHIP_EN = '1') and
238ba3 2 = '0') else '0';
f8eb19 ADDR_KEY22(5) <= '1' when ((ADDR = "1110101") and (CHIP_EN = '1') and
d88ba3 2 = '0') else '0';
85bf9 ADDR_KEY22(6) <= '1' when ((ADDR = "1110110") and (CHIP_EN = '1') and
038ba3 2 = '0') else '0';
fcfa5a 91f558 ADDR_KEY23(0) <= '1' when ((ADDR = "1111000") and (CHIP_EN = '1') and
c18ba3 2 = '0') else '0';
78d725 ADDR_KEY23(1) <= '1' when ((ADDR = "1111001") and (CHIP_EN = '1') and
6467c5 ADDR_KEY23(2) <= '1' when ((ADDR = "1111100") and (CHIP_EN = '1') and
178ba3 2 = '0') else '0';
c345b8 ADDR_KEY23(3) <= '1' when ((ADDR = "1111101") and (CHIP_EN = '1') and
df8ba3 2 = '0') else '0';
c5cf1c ADDR_KEY23(4) <= '1' when ((ADDR = "1111100") and (CHIP_EN = '1') and
bd8ba3 2 = '0') else '0';
78ed61 ADDR_KEY23(5) <= '1' when ((ADDR = "1111101") and (CHIP_EN = '1') and
b78ba3 2 = '0') else '0';
c65d81 ADDR_KEY23(6) <= '1' when ((ADDR = "1111110") and (CHIP_EN = '1') and
348ba3 2 = '0') else '0';
7fa5a 6855356 -----------------------------------
c0b88a end beh;
9c5356 -----------------------------------
7ea5a
`--f530 0001ae1063b8030002 Page 1 of des.vhd

bb097d ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
aa533a -- Author ----------: Tom Vu ---------------------|---------------------|---------------------|---------------------|---------------------|
bf8d8a -- Date ----------: 09/27/97 ---------------------|
b4d6cd -- Description: ... Left and Right 32-bit registers ...

aa7af library ieee;
cada83 use IEEE.std_logic_1164.all;
cada89 use IEEE.std_logic_arith.all;
7ef05 use IEEE.std_logic_unsigned.all;

4df62a -- ---------------------
4d7373 entity MESG is
4a7af5a e1c57e port( .CLK , . . . : in . . std_logic;
363f61 .RST_N , . . : in . . std_logic;
d21689 .START , . . : in . . std_logic;
f4d477 .DONE , . . : in . . std_logic;
e9184d MESSAGE , . . : in . . std_logic_vector(63 downto 0);
ea346b SUBKEY , . . : in . . std_logic_vector(47 downto 0);
b1a256 RESULT , . . : out . . std_logic_vector(63 downto 0);
d737c -- );
a9a5f5a 753a26 end MESG;
daaf5a be625a --
1e41f0 architecture beh of MESG is

c6d25a --
4daf5a c739ea signal IP_KEY , . . : std_logic_vector(63 downto 0);
97e79c signal MESG_LEFT , . . : std_logic_vector(31 downto 0);
c1a87 signal MESG_RIGHT , . . : std_logic_vector(31 downto 0);
023d2 signal NEW_L , . . : std_logic_vector(31 downto 0);
1ea3d signal L , . . : std_logic_vector(31 downto 0);
814c49 signal R , . . : std_logic_vector(31 downto 0);
e1d2c1 signal EXPANDED_R , . . : std_logic_vector(47 downto 0);
73178d signal X_KEY , . . : std_logic_vector(47 downto 0);
b2d20b signal S_OUT , . . : std_logic_vector(31 downto 0);
91e1cda signal FP_IN , . . : std_logic_vector(63 downto 0);
c9d3b signal FP_OUT , . . : std_logic_vector(63 downto 0);
ff907c signal P_IN , . . : std_logic_vector(31 downto 0);
b66db2 signal P_OUT , . . : std_logic_vector(31 downto 0);

daaf5a f214be component EX

5913ef ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
d5f417 ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|
b1e2c6 end component;
10a5f5a cc4e1 component IP
f5c56d port( . . . . . . . . . . . . . ;
86bf1b ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
5e694d ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
f0e2c6 end component;
44af5a d00e26 component FP
36c6d port( . . . . . . . . . . . ;
ed41a9 ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
81ce7 ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
18e2c6 end component;
6ca5f5a 8d6a7a component P
7e0e6d port( . . . . . . . . . ;
e4e2c ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
80f1a6 ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
79e2c6 end component;
21a5f5a 028f9c component S_TABLE
7b96b5 port( .KEY , . . . . . ;
110146 ---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------|---------------------#
32e2c6 end component;
87a5f5a 3b0f89 begin
1aaf5a

Chapter 6: Chip Source Code

Code"
--aaa2 00048aa879b80030002 Page 2 of des.vhd

7d0363 MESSAGE1: EX
8f7097 port map(......
e5c895 => EX_IN ......... => R,
e4f1b2 => EX_OUT ......... => EXPANDED_R);
6eaf5a

de647d MESSAGE2: IP
b87097 port map(......
f358db => IP_IN ......... => MESSAGE,
284c3d => IP_OUT ......... => IP_KEY);
ceaf5a

0a1b74 MESSAGE3: S_TABLE
dd7097 port map(......
05527c => KEY ......... => X_KEY,
0a7ff2 => S_OUT ......... => S_OUT);
d6af5a

fadc73 MESSAGE4: P
437097 port map(......
14299f => P_IN ......... => IP_IN,
5cf3c6 => P_OUT ......... => P_OUT);
0afaf5a

afaf5a
e8326b MESSAGE5: FP
97097 port map(......
b191f5 => FP_IN ......... => FP_IN,
c8be19 => FP_OUT ......... => FP_OUT);
5afaf5a

b2af5a

cc625a -- Split_to_LEFT_and_RIGHT: process(IP_KEY)
39baf5a

339acc -- begin
cdd360 => for i in 0 to 31 loop
bd964e => ..... MESG_RIGHT(i) => IP_KEY(i);
5f6b1b => ..... MESG_LEFT(i) => IP_KEY(i+52);
913689 => end loop;
0d55d2 => end process Split_to_LEFT_and_RIGHT;
74af5a

191a19 MESG_RIGHT <= IP_KEY(31 downto 0);
7a284c MESG_LEFT <= IP_KEY(63 downto 32);
b3625a -- --

726ec6 L_AND_R_REG_PR: process(RST_N,CLK)
7e625a -- --

080f89 begin
ac6118 if RST_N = '0' then
8ce37f => L <= (others => '0');
38ab45 => R <= (others => '0');
a68bd8 elsif CLK'event and CLK = '1' then
560a81 if (START = '1') then
e40d0d => L <= MESG_LEFT;
ff2134 => R <= MESG_RIGHT;
f9def1 else.
f8afcf => L <= R;
0a1c45 => R <= NEW_L;
bd4f0b end if;
addf0b end if;
9baf5a

6b5480 end process L_AND_R_REG_PR;
a5af5a

3c625a -- --

1d9726 KEY_XOR_PR: process(SUBKEY,EXPANDED_R)
66e625a -- --

44e8af begin
0b13e9 => for i in 0 to 47 loop
e8bd9 => ..... X_KEY(i) <= SUBKEY(i) xor EXPANDED_R(i);
737aa9 => end loop;
2fa96e end process KEY_XOR_PR;
2eaf5a

9f5de9 L_XOR_PR: process(L,P_OUT)
6e625a -- --

db0f89 begin
48d72a => for i in 0 to 31 loop


-- In des.vhd

1e3971 > ....NEW_L(i) <= L(i) xor P_OUT(i);
5a7a9 > end loop;
40a92f end process L_XOR_PR;
aaf5a
58e52a -- Combine final L and R to FP
19625a
74c45a > FP_IN <= NEW_L(31 downto 0) & R(31 downto 0);
5af5a
ad625a
76a4c0 RESULT_PR: process(RST_N,CLK)
86625a begin
04018 if RST_N = '0' then
e9018b 
be369 > RESULT <= (others => '0');
b684bd else if CLK'event and CLK = '1' then
9a4e0a if (DONE = '1') then
0e3352f > RESULT <= FP_OUT;
ecdf0b end if;
5edf0b end if;
61af5a
d402a3 end process RESULT_PR;
69a5a
05625a -- end beh;
b9625a
02af5a
8c7fa5 library ieee;
17f1e9 use IEEE.std_logic_1164.all;
5d5a83 use IEEE.std_logic_arith.all;
8ae105 use IEEE.std_logic_unsigned.all;
78625a --
d86f49 entity DES is
3af5a
48c57e port( .CLK > .... : in .... std_logic;
22f61 > .... RST_N > .... : in .... std_logic;
1e1689 > .... START > .... : in .... std_logic;
921069 > .... MESSAGE > .... : in .... std_logic_vector(63 downto 0);
1be2a > .... KEY > .... : in .... std_logic_vector(55 downto 0);
c319f6 > .... DONE > .... : out .... std_logic;
858cf6 > .... CNT > .... : out .... std_logic_vector(4 downto 0);
65be3 > .... DES_OUT > .... : out .... std_logic_vector(63 downto 0);
a2737c ....);
2af5a
8ccbd8 end DES;
81af5a
fa625a --
18f4f4 architecture beh of DES is
31625a --
3bf4fd signal SUBKEY : std_logic_vector(47 downto 0);
6b1864 signal DONE_BAK : std_logic;
42af5a
74c4f4 component MESS
f9c57e port( .CLK > .... : in .... std_logic;
063f61 > .... RST_N > .... : in .... std_logic;
051689 > .... START > .... : in .... std_logic;
be4f77 > .... DONE > .... : in .... std_logic;
f11069 > .... MESSAGE : in .... std_logic_vector(63 downto 0);
4c338a > .... SUBKEY : in .... std_logic_vector(63 downto 0);
7ea256 > .... RESULT : in .... std_logic_vector(63 downto 0);
ca737c ....);
79e26e end component;
2daf5a
cddf define component KEY_GEN
15d6c7 port( .CLK : in .... std_logic;
1e32d > .... RST_N : in .... std_logic;
667f2d > .... START : in .... std_logic;
46b9a > .... KEY_IN : in .... std_logic_vector(55 downto 0);
cb6da8 > .... DONE : out .... std_logic;
19a5f8 > .... CNT : out .... std_logic_vector(4 downto 0);
bb845e > .... KEY_OUT : out .... std_logic_vector(47 downto 0));
c6e2c6 end component;
--381d 00180ac228180030002 Page 4 of des.vhd

e0af5a 4a0f89 begin
e0af5a 554a2f DES1: MSEG
2e1940 port map ( ... 1392d2 => CLK, 49c81c ... RST_N => RST_N, 223042 ... START => START, fef4ab ... DONE => DONE_BAK, 1138b3 ... MESSAGE => MESSAGE, f157a1 ... SUBKEY => SUBKEY, 160e22 ... RESULT => DES_OUT faa415 port map ( ... );
b30759 DES2: KEY_GEN
857097 port map ( ... 139892 ... CLK => CLK, f28c8a ... RST_N => RST_N, c6b76a ... START => START, 22bffc ... KEY_IN => KEY, 5045c8 ... DONE => DONE_BAK, 5b11ad ... CNT => CNT, 4ac932 ... KEY_OUT => SUBKEY 130886 port map ( ... ); 15af5a 834d0f DONE <= DONE_BAK; 68af5a c9b08a end beh; 80af5a
-- 1918 0000b92e30e480030003 Page 1 of des_ctl.vhd

bb979d -------------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|

e7625a

9cde87 -- AUTHOR .............: TOM VU

0773db -- DATE ...............: 10/15/97

e56087 -- LIBRARY ............: std_logic_textio.all

8178f4 -- FILE ...............: des_ctl.vhd

1b997d -- -------------------:-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|

8c5356 -- -------------------:-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|

977afe (library ieee;

c111e9 use IEEE.std_logic_1164.all;
b2da83 use IEEE.std_logic_arith.all;
bde105 use IEEE.std_logic_unsigned.all;
f05c9a use ieee.std_logic_textio.all;

9f8f5a b03876 entity CTL is

3a6f5a port ( .CLK .............: in .... std_logic;

f42e77 ......... RST_N ............: in .... std_logic;

99d3de ...... START .............: in .... std_logic;

8ab3d7 ....... DECR .............: in .... std_logic;

ef539c ...... DECR_INT ..........: out .... std_logic;

d148b9 ...... READ_EN ............: out .... std_logic;

d4e72f ...... START_INT ..........: out .... std_logic;

b5871b ...... MESSAGE ..........: out .... std_logic_vector(63 downto 0);

5eb5fb ...... KEY .................: out .... std_logic_vector(55 downto 0);

b06cd6 ...... DATA ..............: inout std_logic_vector(31 downto 0);

e273fc ...... :

b1af5a 88ebc end CTL;

0daf5a

3e625a architecture BEH of CTL is

e3625a

27a6f5a b9c8c3 signal CNT16 ............: std_logic_vector(3 downto 0);

44beb8 signal MSG0 .............: std_logic_vector(31 downto 0);

716d50 signal MSG1 .............: std_logic_vector(31 downto 0);

905b36 signal KEY0 .............: std_logic_vector(31 downto 0);

1428a9 signal KEY1 .............: std_logic_vector(23 downto 0);

7eb0f7 signal OUT0 .............: std_logic_vector(31 downto 0);

905b63 signal OUT1 .............: std_logic_vector(31 downto 0);

b0c721 signal DATA_BAK ..........: std_logic_vector(31 downto 0);

fd3ce5 signal START_INT_D .... : std_logic;

7ca5f5a d50f89 begin

37625a 88c6b7 process(CLK,RST_N);

3e6f5a

230f89 begin

4b1801 .......... if RST_N = '0' then

4ac97a .......... -- CNT0 <= "0000";

89f45b .......... START_INT <= '0';

e9d3d2 .......... if DECR_INT <= '0' then

ed9e8f .......... else if CLK'event and CLK = '1' then

b59193 .......... -- START_INT <= START_INT_D;

5eb749 .......... if CNT16 = 4 then

419236 .......... -- DECR_INT <= DECR;

d1b24d .......... end if;

863598 .......... if START = '1' then

edc5c1 .......... CNT16 <= "0001";

ef593c .......... else

b936d .......... -- CNT16 <= CNT16 + 1;

8101cd .......... end if;

fcb985 .......... end if;

7cc5800 end process;

d8aaf5a

55625a

542e63 KEY <= KEY1 & KEY0;

3998b6 MESSAGE <= MSG1 & MSG0;

8daf5a

b9875a START_INT_D <= '1' when CNT16 = 4 else '0';
Chapter 6: Chip Source Code

--5868 0006ca077a80030003 Page 2 of des_ctl.vhd

3b625a 6-13

18 REG_IN_PR: process (RST_N, CLK)

f0625a

b0f89 begin

63618 if RST_N = '0' then

f8215b 6-13

b188b6 if MSG0 <= (others => '0') then

83e9c5 6-13

b84011 if KEY0 <= (others => '0') then

b60502 elsif CLK'event and CLK = '0' then

7e8081 6-13
case CNT16 is

2a0cfb 6-13

975aa4 6-13

bb9705 6-13

d56f35 6-13

a2932e 6-13

b26b0e 6-13

57896f 6-13

e8f94a 6-13

283cff 6-13

4634e8 6-13

cf9517 6-13

39df0b end if;

c3af5a

2c7f1c end process REG_IN_PR;

636bf6

5d7518 MESSAGE_OUT_P: process (CNT16, DES_OUT)

def0f89 begin

1bf081 6-13
case CNT16 is

76eac3 6-13

dc5734 6-13

b475bf 6-13

9bb995 6-13

e91fd3 6-13

5e75bf 6-13

3e3cfc 6-13

322d30 6-13

9b6904 6-13

cf9517 6-13

885890 end process;

28d83c

c66687 DATA <= DATA_BAK;

645c73 end BEH;

aa625a

77f7af library ieee;

4017e9 use IEEE.std_logic_1164.all;

9dabf8 use IEEE.std_logic_arith.all;

47b105 use IEEE.std_logic_unsigned.all;

ab5ca9 use ieee.std_logic_textio.all;

d3f403 use std.textio.all;

8da5fa

6af5a

79bf60 entity DES_CTL is:

d5ec7a port (CLK : in std_logic;

ef2e77 RST_N : in std_logic;

e83d3e START : in std_logic;

67bd37 DECR : in std_logic;

be6d3e START : in std_logic;

b5b347 DECR : in std_logic;

4ee222 component DES:

15ec7a port (CLK : in std_logic;

b5b347 architecture beh of DES_CTL is

7bf08 c7af5a
Chapter 6: Chip Source Code

--4c0d 001891375c38003003 Page 3 of des_ctl.vhd

#6095 MESSAGE : : in std_logic_vector(63 downto 0);
#b57b4f KEY : : in std_logic_vector(55 downto 0);
#389930 DONE : out std_logic;
#443fe1 DES_OUT : out std_logic_vector(63 downto 0)
#4373c ;

14af5a cae2c6 end component;
9c9f5a 89c2b6 component CTL:
2aec7a port(• CLK • • RST_N • • START • • DECR • • MESSAGE • • KEY • • DATA
start end component;
9c9f5a
89c2b6 component CTL:
2aec7a port(• CLK • • RST_N • • START • • DECR • • MESSAGE • • KEY • • DATA
b2e2c6 end component;
6af5a e1157a signal START_INT : : std_logic;
ce4d61 signal DECR_INT : : std_logic;
2af53c signal MESSAGE : : std_logic_vector(63 downto 0);
3c9af4 signal KEY : : std_logic_vector(55 downto 0);
0e7fd8 signal DES_OUT : : std_logic_vector(63 downto 0);
33af5a ---------------------------------------------
800f89 begin
1da5a
688c8c DES_CTL1 : DES:
58d9d1 port map:
01d07d CLK => CLK,
182dd5 RST_N => RST_N,
d44a53 START => START_INT,
8363b8 DECR => DECR_INT,
68fb6c MESSAGE => MESSAGE,
dff105 KEY => KEY,
e8de19 DONE => DONE,
9f5f5f DES_OUT => DES_OUT
3e627f ;
8caf5a 0be84d DES_CTL2 : CTL
c215ed port map(• CLK => CLK,
f5a6a • RST_N => RST_N,
2dc2ff START => START,
c176c DECR => DECR,
9fd115 DES_OUT => DES_OUT,
a10875 DECR_INT => DECR_INT,
a721c4 READ_EN => READ_EN,
7be98 START_INT => START_INT,
cf2a6b MESSAGE => MESSAGE,
2f2a6a KEY => KEY,
c950dc DATA => DATA
e573c ;
80af5a 89b08a end beh;
Library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity EX is
  port (EX-IN: std_logic_vector(31 downto 0);
        EX-OUT: std_logic_vector(47 downto 0));
end EX;

architecture beh of EX is
  use IEEE.numeric_std.all;
  use IEEE.numeric_std_unsigned.all;

  -- 32-bit Left and Right registers
  subtype small_integer is INTEGER range 0 to 31;
  type EX-TYPE is array(0 to 47) of small_integer;

  signal EX-TABLE: EX-TYPE;

begin
  EX-TABLE <= (31, 0, 1, 2, 3, 4,
                5, 6, 7, 8, 9, 10, 11, 12,
                13, 14, 15, 16, 17, 18, 19, 20,
                21, 22, 23, 24, 25, 26, 27, 28,
                29, 30, 31, 0);

  EX-PR: process(EX-IN,EX-TABLE)
  begin
    for i in 0 to 47 loop
      EX-OUT(i) <= EX-IN(EX-TABLE(i));
    end loop;
  end process EX-PR;

end beh;
--8e68 00126a90e980030005 Page 1 of fp.vhd

bb997d ----------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
aa533a -- Author       -- Tom Vu                     ------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
bf8d0a -- Date         -- 09/27/97                    ------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
b4d6cd -- Description  -- Left and Right 32-bit registers ------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
a58629a -- ------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|------------------|
aa7fa5a
5211e9 use IEEE.std_logic_1164.all;
cada8b use IEEE.std_logic_arith.all;
77e105 use IEEE.std_logic_unsigned.all;
05af5a
59af5a
7b625a
98af5a
08625a
14962b entity FP is
87af5a
bbcc6d port(
9a41a9       FP_IN          : in     std_logic_vector(63 downto 0);
b43f8e       FP_OUT         : out    std_logic_vector(63 downto 0);
3d737c       );
31af5a
a7b2d6 end FP;
79625a
8e7c77 architecture beh of FP is
70625a
7f7bd6 subtype small_integer is INTEGER range 0 to 63;
6b988c type FP_TYPE is array(0 to 63) of small_integer;
92af5a
53dabe signal FP_TABLE : FP_TYPE;
62af5a
800f89 begin
21af5a
ed66ec FP_TABLE <= (57,49,41,33,25,17,9,1,
fa1781       ,59,51,43,35,27,19,11,3,
7b15ff       ,61,53,45,37,29,21,13,5,
f2d996       ,63,55,47,39,31,23,15,7,
84783c       ,56,48,40,32,24,16,8,0,
795d3b       ,58,50,42,34,26,18,10,2,
9abf45       ,60,52,44,36,28,20,12,4,
36e3e3       ,62,54,46,38,30,22,14,6);
f0af5a
fdaf5a
71625a
3c3fd4 FP_PR: process(FP_TABLE,FP_IN) fb625a
dc0f89 begin 623e98 for i in 0 to 63 loop 91679a       FP_OUT(FP_TABLE(i)) <= FP_IN(i);
9636d5       end loop;
3ea33a end process FP_PR;
61625a
70b08a end beh;
26af5a
Chapter 6: Chip Source Code
Author: Tom Vu
Date: 09/27/97

Description:

• Left and Right 32-bit registers

library ieee;
use IEEE.std_logic_1164.all;

use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity IP is
  port(
    IP-IN : in std_logic_vector(63 downto 0);
    IP-OUT : out std_logic_vector(63 downto 0)
  );
end IP;

architecture beh of IP is
  subtype small_integer is INTEGER range 0 to 63;
  type IP-TYPE is array(0 to 63) of small_integer;
  signal IP-TABLE : IP-TYPE;

begin
  IP-TABLE <= (39, 7, 47, 15, 55, 23, 63, 31, 38, 6, 46, 14, 54, 22, 62, 30, 37, 5, 45, 13, 53, 21, 61, 29, 36, 4, 44, 12, 52, 20, 60, 28, 35, 3, 43, 11, 51, 19, 59, 27, 34, 2, 42, 10, 50, 18, 58, 26, 33, 1, 41, 9, 49, 17, 57, 25, 32, 0, 40, 8, 48, 16, 56, 24);

  for i in 0 to 63 loop
    IP-OUT(IP-TABLE(i)) <= IP-IN(i);
  end loop;
end process IP-PR;

begin
  for i in 0 to 63 loop
    IP-OUT(IP-TABLE(i)) <= IP-IN(i);
  end loop;
end process IP-PR;

end beh;
-- c4cf 000de4ae3178030007 Page 1 of key_gen.vhd

Chapter 6: Chip Source Code

bb997d ----------------------------------------|--------|--------|--------|--------|--------|--------|--------|
d8b1e9 -- Author: Tom Vu------------------------
e8f80a -- Date: 09/27/97 ------------------------
e66b31 -- Description: Generate Schedule Keys to be used by Function
da47f7 -- Function: 2 rings of 28 bits each will shift left or right by 1 or
3efb57 -- 2 positions depends on ENCR/DECR and counter
37625a ----

ae7af5 library ieee;
c11ef9 use IEEE.std_logic_1164.all;
d4da83 use IEEE.std_logic_arith.all;
e6e105 use IEEE.std_logic_unsigned.all;
1fafa5a
ae7af5a
d625a ----
9e1267 entity KEY_GEN is
20af5a 1284ad port( RST_N: in std_logic;
33e26 --- START: in std_logic;
5f77c -- DECR: in std_logic;
5fa4d > KEY_IN: in std_logic_vector(55 downto 0);
2dbaf1 KEY_OUT: out std_logic;
a2970a CNT: out std_logic_vector(4 downto 0);
1fe70b KEY_OUT: out std_logic_vector(47 downto 0);
6c737c ---);
ccaf5a adf875 end KEY_GEN;
20af5a
af625a ---
667511 architecture beh of KEY_GEN is
fb625a ---
c6af5a 2f801d component PC1;
d114be port( KEY_IN: in std_logic_vector(55 downto 0);
33e01d KEY_OUT: out std_logic_vector(55 downo 0)
7d73c ---);
a6e2c6 end component;
fbaf5a
ac71c8 component PC2
bf14be port( KEY_IN: in std_logic_vector(55 downto 0);
b4e70b KEY_OUT: out std_logic_vector(47 downto 0)
7737c ---);
20e2c6 end component;
3baf5a dd4405 signal cnt16: std_logic_vector(4 downto 0);
66f2bc signal PC1.KEY: std_logic_vector(55 downto 0);
9f5f49 signal PC1.KEY_C: std_logic_vector(27 downto 0);
87dea6 signal PC1.KEY_D: std_logic_vector(27 downto 0);
742bd5 signal KEY_REG_C: std_logic_vector(27 downto 0);
5e88f2 signal KEY_REG_D: std_logic_vector(27 downto 0);
3c2ca8 signal KEY_REG: std_logic_vector(55 downto 0);
9c08ab signal SHIFT1: std_logic;
27af5a
3baf5a 5df89 begin
f3af5a
66625a
b088e6 -- Permutation Choice #1
16625a
20af5a 82d755 PC1: PC1 port map(KEY_IN => KEY_IN, KEY_OUT => PC1.KEY);
e7af5a 9e625a 1d232d Split_to_C_and_D: process(PC1.KEY)
6b625a
59f89 begin
b4e58 for i in 0 to 27 loop
584cd9 ---PC1.KEY_D(i) <= PC1.KEY(i);
571758 ---PC1.KEY_C(i) <= PC1.KEY(i+28);
de7aa9 end loop;
6-19

```vhdl
--172b 000fd12a97a8030007 Page 2 of key_gen.vhd

325890 end process;
bc625a

if CASO_P process

50625a begin

190f89 if RST_N = '0' then

614864 6472d

581793

9687cc

46e192

14b77

36d0c1

end if;

79a5a

617b70

49ed6a

c692536

96e62e

99d0c1

end if;

5857d0

end process;

c1625a

9bbc60 COUNTER16_P: process(CLK, RST_N);

c0c625a begin

14b89 if RST_N = '0' then

534864 6dc8a0

23bdcf

d76769

330000

65e192

635a7

6d0d1c

end if;

5857d0 end process;

05625a

442b00 KEY_GEN_REG_P: process(CLK, RST_N);

85625a

a60f89 begin

85a5a

b8def7

bd44a

ed927a

8fa6a4

b940e9

5c49cf

d55710

4d3e6c

734ec4

6ed797

6193d0

c747e3c

9bb0b0

a97edc

df73f5

689d58

47d271

71358

c6809e

a31e3f

a97e34

87e192

77e90

718558

72d0b6

a10d1c

b99c3

785175

dataf5a
```

---2e6c 001be95937580030007 Page 3 of key_gen.vhd

325890 end process;
bc625a -- ---------------------------------------------------------------
44e546 -- Combine final C and D to KEY_REG
cf625a -- ---------------------------------------------------------------
7b2776 > KEY_REG <= KEY_REG_C(27 downto 0) & KEY_REG_D(27 downto 0);
02625a -- ---------------------------------------------------------------
b8a28e -- Permutation Choice #2
a2625a -- ---------------------------------------------------------------
e5af5a > PC_2: PC2 port map (KEY_IN => KEY_REG,KEY_OUT => KEY_OUT);
d4105b -- ---------------------------------------------------------------
2daf5a 1a625a -- ---------------------------------------------------------------
8dd318 CNT <= CNT16;
10b08a end beh;---------------------------------------------------------------
be625a -- ---------------------------------------------------------------
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity MUX256 is
  port(
    SHIFT_OUT : in std_logic_vector(7 downto 0);
    PT VECTOR : in std_logic_vector(255 downto 0);
    BIT_MUX : out std_logic);
end MUX256;

architecture beh of MUX256 is
begin
DECODER PR: process(SHIFT_OUT, PT VECTOR) begin
  i : integer;
  when 0 => BIT_MUX <= PT VECTOR(0);
  when 1 => BIT_MUX <= PT VECTOR(1);
  when 2 => BIT_MUX <= PT VECTOR(2);
  when 3 => BIT_MUX <= PT VECTOR(3);
  when 4 => BIT_MUX <= PT VECTOR(4);
  when 5 => BIT_MUX <= PT VECTOR(5);
  when 6 => BIT_MUX <= PT VECTOR(6);
  when 7 => BIT_MUX <= PT VECTOR(7);
  when 8 => BIT_MUX <= PT VECTOR(8);
  when 9 => BIT_MUX <= PT VECTOR(9);
  when 10 => BIT_MUX <= PT VECTOR(10);
  when 11 => BIT_MUX <= PT VECTOR(11);
  when 12 => BIT_MUX <= PT VECTOR(12);
  when 13 => BIT_MUX <= PT VECTOR(13);
  when 14 => BIT_MUX <= PT VECTOR(14);
  when 15 => BIT_MUX <= PT VECTOR(15);
  when 16 => BIT_MUX <= PT VECTOR(16);
  when 17 => BIT_MUX <= PT VECTOR(17);
  when 18 => BIT_MUX <= PT VECTOR(18);
  when 19 => BIT_MUX <= PT VECTOR(19);
  when 20 => BIT_MUX <= PT VECTOR(20);
  when 21 => BIT_MUX <= PT VECTOR(21);
  when 22 => BIT_MUX <= PT VECTOR(22);
  when 23 => BIT_MUX <= PT VECTOR(23);
  when 24 => BIT_MUX <= PT VECTOR(24);
  when 25 => BIT_MUX <= PT VECTOR(25);
  when 26 => BIT_MUX <= PT VECTOR(26);
  when 27 => BIT_MUX <= PT VECTOR(27);
  when 28 => BIT_MUX <= PT VECTOR(28);
  when 29 => BIT_MUX <= PT VECTOR(29);
end process;
end architecture;
### Chapter 6: Chip Source Code

- **e5c6 0004c472018030000 Page 2 of mux256.vhd**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>292479</td>
<td>when 30 =&gt; BIT_MUX &lt;= PT_VECTOR(30);</td>
</tr>
<tr>
<td>257a60</td>
<td>when 31 =&gt; BIT_MUX &lt;= PT_VECTOR(31);</td>
</tr>
<tr>
<td>697176</td>
<td>when 32 =&gt; BIT_MUX &lt;= PT_VECTOR(32);</td>
</tr>
<tr>
<td>a06f05</td>
<td>when 33 =&gt; BIT_MUX &lt;= PT_VECTOR(33);</td>
</tr>
<tr>
<td>b3e5f76</td>
<td>when 34 =&gt; BIT_MUX &lt;= PT_VECTOR(34);</td>
</tr>
<tr>
<td>5983f9</td>
<td>when 35 =&gt; BIT_MUX &lt;= PT_VECTOR(35);</td>
</tr>
<tr>
<td>3e127f9</td>
<td>when 36 =&gt; BIT_MUX &lt;= PT_VECTOR(36);</td>
</tr>
<tr>
<td>a93816</td>
<td>when 37 =&gt; BIT_MUX &lt;= PT_VECTOR(37);</td>
</tr>
<tr>
<td>6a8077</td>
<td>when 38 =&gt; BIT_MUX &lt;= PT_VECTOR(38);</td>
</tr>
<tr>
<td>a101e8</td>
<td>when 39 =&gt; BIT_MUX &lt;= PT_VECTOR(39);</td>
</tr>
<tr>
<td>1b589a</td>
<td>when 40 =&gt; BIT_MUX &lt;= PT_VECTOR(40);</td>
</tr>
<tr>
<td>a96e95</td>
<td>when 41 =&gt; BIT_MUX &lt;= PT_VECTOR(41);</td>
</tr>
<tr>
<td>a9e395</td>
<td>when 42 =&gt; BIT_MUX &lt;= PT_VECTOR(42);</td>
</tr>
<tr>
<td>be3a1a</td>
<td>when 43 =&gt; BIT_MUX &lt;= PT_VECTOR(43);</td>
</tr>
<tr>
<td>872695</td>
<td>when 44 =&gt; BIT_MUX &lt;= PT_VECTOR(44);</td>
</tr>
<tr>
<td>d4a69a</td>
<td>when 45 =&gt; BIT_MUX &lt;= PT_VECTOR(45);</td>
</tr>
<tr>
<td>569d99</td>
<td>when 46 =&gt; BIT_MUX &lt;= PT_VECTOR(46);</td>
</tr>
<tr>
<td>5e4415</td>
<td>when 47 =&gt; BIT_MUX &lt;= PT_VECTOR(47);</td>
</tr>
<tr>
<td>2d5a48</td>
<td>when 48 =&gt; BIT_MUX &lt;= PT_VECTOR(48);</td>
</tr>
<tr>
<td>ed799b</td>
<td>when 49 =&gt; BIT_MUX &lt;= PT_VECTOR(49);</td>
</tr>
<tr>
<td>4e24bc</td>
<td>when 50 =&gt; BIT_MUX &lt;= PT_VECTOR(50);</td>
</tr>
<tr>
<td>63f3d33</td>
<td>when 51 =&gt; BIT_MUX &lt;= PT_VECTOR(51);</td>
</tr>
<tr>
<td>3e9f3b3</td>
<td>when 52 =&gt; BIT_MUX &lt;= PT_VECTOR(52);</td>
</tr>
<tr>
<td>da4635</td>
<td>when 53 =&gt; BIT_MUX &lt;= PT_VECTOR(53);</td>
</tr>
<tr>
<td>cc5a53</td>
<td>when 54 =&gt; BIT_MUX &lt;= PT_VECTOR(54);</td>
</tr>
<tr>
<td>8d833c</td>
<td>when 55 =&gt; BIT_MUX &lt;= PT_VECTOR(55);</td>
</tr>
<tr>
<td>4e1be1c</td>
<td>when 56 =&gt; BIT_MUX &lt;= PT_VECTOR(56);</td>
</tr>
<tr>
<td>1d392a</td>
<td>when 57 =&gt; BIT_MUX &lt;= PT_VECTOR(57);</td>
</tr>
<tr>
<td>57d8a2</td>
<td>when 58 =&gt; BIT_MUX &lt;= PT_VECTOR(58);</td>
</tr>
<tr>
<td>21012d</td>
<td>when 59 =&gt; BIT_MUX &lt;= PT_VECTOR(59);</td>
</tr>
<tr>
<td>f6a066</td>
<td>when 60 =&gt; BIT_MUX &lt;= PT_VECTOR(60);</td>
</tr>
<tr>
<td>727959</td>
<td>when 61 =&gt; BIT_MUX &lt;= PT_VECTOR(61);</td>
</tr>
<tr>
<td>7e1bd9</td>
<td>when 62 =&gt; BIT_MUX &lt;= PT_VECTOR(62);</td>
</tr>
<tr>
<td>96c256</td>
<td>when 63 =&gt; BIT_MUX &lt;= PT_VECTOR(63);</td>
</tr>
<tr>
<td>e7de9d</td>
<td>when 64 =&gt; BIT_MUX &lt;= PT_VECTOR(64);</td>
</tr>
<tr>
<td>210756</td>
<td>when 65 =&gt; BIT_MUX &lt;= PT_VECTOR(65);</td>
</tr>
<tr>
<td>8c6560</td>
<td>when 66 =&gt; BIT_MUX &lt;= PT_VECTOR(66);</td>
</tr>
<tr>
<td>9fbc59</td>
<td>when 67 =&gt; BIT_MUX &lt;= PT_VECTOR(67);</td>
</tr>
<tr>
<td>985cc8</td>
<td>when 68 =&gt; BIT_MUX &lt;= PT_VECTOR(68);</td>
</tr>
<tr>
<td>598547</td>
<td>when 69 =&gt; BIT_MUX &lt;= PT_VECTOR(69);</td>
</tr>
<tr>
<td>bc5cf0</td>
<td>when 70 =&gt; BIT_MUX &lt;= PT_VECTOR(70);</td>
</tr>
<tr>
<td>5b057f</td>
<td>when 71 =&gt; BIT_MUX &lt;= PT_VECTOR(71);</td>
</tr>
<tr>
<td>dd67ff</td>
<td>when 72 =&gt; BIT_MUX &lt;= PT_VECTOR(72);</td>
</tr>
<tr>
<td>49be06</td>
<td>when 73 =&gt; BIT_MUX &lt;= PT_VECTOR(73);</td>
</tr>
<tr>
<td>b7b70f</td>
<td>when 74 =&gt; BIT_MUX &lt;= PT_VECTOR(74);</td>
</tr>
<tr>
<td>3a1910</td>
<td>when 75 =&gt; BIT_MUX &lt;= PT_VECTOR(75);</td>
</tr>
<tr>
<td>62c07f</td>
<td>when 76 =&gt; BIT_MUX &lt;= PT_VECTOR(76);</td>
</tr>
<tr>
<td>e128cf</td>
<td>when 77 =&gt; BIT_MUX &lt;= PT_VECTOR(77);</td>
</tr>
<tr>
<td>b2f961</td>
<td>when 78 =&gt; BIT_MUX &lt;= PT_VECTOR(78);</td>
</tr>
<tr>
<td>b05910</td>
<td>when 79 =&gt; BIT_MUX &lt;= PT_VECTOR(79);</td>
</tr>
<tr>
<td>56800f</td>
<td>when 80 =&gt; BIT_MUX &lt;= PT_VECTOR(80);</td>
</tr>
<tr>
<td>9562e8</td>
<td>when 81 =&gt; BIT_MUX &lt;= PT_VECTOR(81);</td>
</tr>
<tr>
<td>ca35b9</td>
<td>when 82 =&gt; BIT_MUX &lt;= PT_VECTOR(82);</td>
</tr>
<tr>
<td>a427f7f</td>
<td>when 83 =&gt; BIT_MUX &lt;= PT_VECTOR(83);</td>
</tr>
<tr>
<td>a0fe90</td>
<td>when 84 =&gt; BIT_MUX &lt;= PT_VECTOR(84);</td>
</tr>
<tr>
<td>ea9c10</td>
<td>when 85 =&gt; BIT_MUX &lt;= PT_VECTOR(85);</td>
</tr>
<tr>
<td>79459f</td>
<td>when 86 =&gt; BIT_MUX &lt;= PT_VECTOR(86);</td>
</tr>
<tr>
<td>79a5e8</td>
<td>when 87 =&gt; BIT_MUX &lt;= PT_VECTOR(87);</td>
</tr>
<tr>
<td>207c61</td>
<td>when 88 =&gt; BIT_MUX &lt;= PT_VECTOR(88);</td>
</tr>
<tr>
<td>b22536</td>
<td>when 89 =&gt; BIT_MUX &lt;= PT_VECTOR(89);</td>
</tr>
<tr>
<td>c6fcb9</td>
<td>when 90 =&gt; BIT_MUX &lt;= PT_VECTOR(90);</td>
</tr>
<tr>
<td>929e39</td>
<td>when 91 =&gt; BIT_MUX &lt;= PT_VECTOR(91);</td>
</tr>
<tr>
<td>a247b6</td>
<td>when 92 =&gt; BIT_MUX &lt;= PT_VECTOR(92);</td>
</tr>
<tr>
<td>ca5b32</td>
<td>when 93 =&gt; BIT_MUX &lt;= PT_VECTOR(93);</td>
</tr>
<tr>
<td>9682b6</td>
<td>when 94 =&gt; BIT_MUX &lt;= PT_VECTOR(94);</td>
</tr>
<tr>
<td>24e036</td>
<td>when 95 =&gt; BIT_MUX &lt;= PT_VECTOR(95);</td>
</tr>
<tr>
<td>2939b0</td>
<td>when 96 =&gt; BIT_MUX &lt;= PT_VECTOR(96);</td>
</tr>
<tr>
<td>87d928</td>
<td>when 97 =&gt; BIT_MUX &lt;= PT_VECTOR(97);</td>
</tr>
<tr>
<td>db0a07</td>
<td>when 98 =&gt; BIT_MUX &lt;= PT_VECTOR(98);</td>
</tr>
<tr>
<td>70a5fa</td>
<td>when 99 =&gt; BIT_MUX &lt;= PT_VECTOR(99);</td>
</tr>
<tr>
<td>4c6b58</td>
<td>when 100 =&gt; BIT_MUX &lt;= PT_VECTOR(100);</td>
</tr>
</tbody>
</table>
Chapter 6: Chip Source Code

Page 3 of mux256.vhd

```
--ac24 000bc20ce88030008

1f0b 1d0a when 101 => BIT_MUX <= PT VECTOR(101);
1ed3 1d0a when 102 => BIT_MUX <= PT VECTOR(102);
1a08 1d0a when 103 => BIT_MUX <= PT VECTOR(103);
1a08 1d0a when 104 => BIT_MUX <= PT VECTOR(104);
1a08 1d0a when 105 => BIT_MUX <= PT VECTOR(105);
1a08 1d0a when 106 => BIT_MUX <= PT VECTOR(106);
1a08 1d0a when 107 => BIT_MUX <= PT VECTOR(107);
1a08 1d0a when 108 => BIT_MUX <= PT VECTOR(108);
1a08 1d0a when 109 => BIT_MUX <= PT VECTOR(109);
1a08 1d0a when 110 => BIT_MUX <= PT VECTOR(110);
1a08 1d0a when 111 => BIT_MUX <= PT VECTOR(111);
1a08 1d0a when 112 => BIT_MUX <= PT VECTOR(112);
1a08 1d0a when 113 => BIT_MUX <= PT VECTOR(113);
1a08 1d0a when 114 => BIT_MUX <= PT VECTOR(114);
1a08 1d0a when 115 => BIT_MUX <= PT VECTOR(115);
1a08 1d0a when 116 => BIT_MUX <= PT VECTOR(116);
1a08 1d0a when 117 => BIT_MUX <= PT VECTOR(117);
1a08 1d0a when 118 => BIT_MUX <= PT VECTOR(118);
1a08 1d0a when 119 => BIT_MUX <= PT VECTOR(119);
1a08 1d0a when 120 => BIT_MUX <= PT VECTOR(120);
1a08 1d0a when 121 => BIT_MUX <= PT VECTOR(121);
1a08 1d0a when 122 => BIT_MUX <= PT VECTOR(122);
1a08 1d0a when 123 => BIT_MUX <= PT VECTOR(123);
1a08 1d0a when 124 => BIT_MUX <= PT VECTOR(124);
1a08 1d0a when 125 => BIT_MUX <= PT VECTOR(125);
1a08 1d0a when 126 => BIT_MUX <= PT VECTOR(126);
1a08 1d0a when 127 => BIT_MUX <= PT VECTOR(127);
1a08 1d0a when 128 => BIT_MUX <= PT VECTOR(128);
1a08 1d0a when 129 => BIT_MUX <= PT VECTOR(129);
1a08 1d0a when 130 => BIT_MUX <= PT VECTOR(130);
1a08 1d0a when 131 => BIT_MUX <= PT VECTOR(131);
1a08 1d0a when 132 => BIT_MUX <= PT VECTOR(132);
1a08 1d0a when 133 => BIT_MUX <= PT VECTOR(133);
1a08 1d0a when 134 => BIT_MUX <= PT VECTOR(134);
1a08 1d0a when 135 => BIT_MUX <= PT VECTOR(135);
1a08 1d0a when 136 => BIT_MUX <= PT VECTOR(136);
1a08 1d0a when 137 => BIT_MUX <= PT VECTOR(137);
1a08 1d0a when 138 => BIT_MUX <= PT VECTOR(138);
1a08 1d0a when 139 => BIT_MUX <= PT VECTOR(139);
1a08 1d0a when 140 => BIT_MUX <= PT VECTOR(140);
1a08 1d0a when 141 => BIT_MUX <= PT VECTOR(141);
1a08 1d0a when 142 => BIT_MUX <= PT VECTOR(142);
1a08 1d0a when 143 => BIT_MUX <= PT VECTOR(143);
1a08 1d0a when 144 => BIT_MUX <= PT VECTOR(144);
1a08 1d0a when 145 => BIT_MUX <= PT VECTOR(145);
1a08 1d0a when 146 => BIT_MUX <= PT VECTOR(146);
1a08 1d0a when 147 => BIT_MUX <= PT VECTOR(147);
1a08 1d0a when 148 => BIT_MUX <= PT VECTOR(148);
1a08 1d0a when 149 => BIT_MUX <= PT VECTOR(149);
1a08 1d0a when 150 => BIT_MUX <= PT VECTOR(150);
1a08 1d0a when 151 => BIT_MUX <= PT VECTOR(151);
1a08 1d0a when 152 => BIT_MUX <= PT VECTOR(152);
1a08 1d0a when 153 => BIT_MUX <= PT VECTOR(153);
1a08 1d0a when 154 => BIT_MUX <= PT VECTOR(154);
1a08 1d0a when 155 => BIT_MUX <= PT VECTOR(155);
1a08 1d0a when 156 => BIT_MUX <= PT VECTOR(156);
1a08 1d0a when 157 => BIT_MUX <= PT VECTOR(157);
1a08 1d0a when 158 => BIT_MUX <= PT VECTOR(158);
1a08 1d0a when 159 => BIT_MUX <= PT VECTOR(159);
1a08 1d0a when 160 => BIT_MUX <= PT VECTOR(160);
1a08 1d0a when 161 => BIT_MUX <= PT VECTOR(161);
1a08 1d0a when 162 => BIT_MUX <= PT VECTOR(162);
1a08 1d0a when 163 => BIT_MUX <= PT VECTOR(163);
1a08 1d0a when 164 => BIT_MUX <= PT VECTOR(164);
1a08 1d0a when 165 => BIT_MUX <= PT VECTOR(165);
1a08 1d0a when 166 => BIT_MUX <= PT VECTOR(166);
1a08 1d0a when 167 => BIT_MUX <= PT VECTOR(167);
1a08 1d0a when 168 => BIT_MUX <= PT VECTOR(168);
1a08 1d0a when 169 => BIT_MUX <= PT VECTOR(169);
1a08 1d0a when 170 => BIT_MUX <= PT VECTOR(170);
1a08 1d0a when 171 => BIT_MUX <= PT VECTOR(171);
1a08 1d0a when 172 => BIT_MUX <= PT VECTOR(172);
```
| 5b2137 | when 173 => BIT_MUX <= PT_VECTOR(173); |
| 6e3a29 | when 174 => BIT_MUX <= PT_VECTOR(174); |
| 965f10 | when 175 => BIT_MUX <= PT_VECTOR(175); |
| faed9b | when 176 => BIT_MUX <= PT_VECTOR(176); |
| 858e42 | when 177 => BIT_MUX <= PT_VECTOR(177); |
| 33db7f | when 178 => BIT_MUX <= PT_VECTOR(178); |
| 3eb07e | when 179 => BIT_MUX <= PT_VECTOR(179); |
| 152da2 | when 180 => BIT_MUX <= PT_VECTOR(180); |
| 10467b | when 181 => BIT_MUX <= PT_VECTOR(181); |
| bdf410 | when 182 => BIT_MUX <= PT_VECTOR(182); |
| 2191c9 | when 183 => BIT_MUX <= PT_VECTOR(183); |
| 808ad7 | when 184 => BIT_MUX <= PT_VECTOR(184); |
| 64e10e | when 185 => BIT_MUX <= PT_VECTOR(185); |
| 7c5d63 | when 186 => BIT_MUX <= PT_VECTOR(186); |
| 2f36bc | when 187 => BIT_MUX <= PT_VECTOR(187); |
| f06b59 | when 188 => BIT_MUX <= PT_VECTOR(188); |
| 420068 | when 189 => BIT_MUX <= PT_VECTOR(189); |
| 2e6085 | when 190 => BIT_MUX <= PT_VECTOR(190); |
| 3305c5 | when 191 => BIT_MUX <= PT_VECTOR(191); |
| 92b137 | when 192 => BIT_MUX <= PT_VECTOR(192); |
| 4adee2 | when 193 => BIT_MUX <= PT_VECTOR(193); |
| 4c1f0 | when 194 => BIT_MUX <= PT_VECTOR(194); |
| 20aa29 | when 195 => BIT_MUX <= PT_VECTOR(195); |
| 181642 | when 196 => BIT_MUX <= PT_VECTOR(196); |
| bf7d9b | when 197 => BIT_MUX <= PT_VECTOR(197); |
| b5296c | when 198 => BIT_MUX <= PT_VECTOR(198); |
| e84baf | when 199 => BIT_MUX <= PT_VECTOR(199); |
| d3af5a | when 200 => BIT_MUX <= PT_VECTOR(200); |
| 2b9a22 | when 201 => BIT_MUX <= PT_VECTOR(201); |
| dc1f7b | when 202 => BIT_MUX <= PT_VECTOR(202); |
| f94d10 | when 203 => BIT_MUX <= PT_VECTOR(203); |
| c126c9 | when 204 => BIT_MUX <= PT_VECTOR(204); |
| bf3d7f | when 205 => BIT_MUX <= PT_VECTOR(205); |
| db560e | when 206 => BIT_MUX <= PT_VECTOR(206); |
| f4e65 | when 207 => BIT_MUX <= PT_VECTOR(207); |
| 7381bc | when 208 => BIT_MUX <= PT_VECTOR(208); |
| 6fd5c9 | when 209 => BIT_MUX <= PT_VECTOR(209); |
| 4f87b0 | when 210 => BIT_MUX <= PT_VECTOR(210); |
| 0bd185 | when 211 => BIT_MUX <= PT_VECTOR(211); |
| 1eb5c5 | when 212 => BIT_MUX <= PT_VECTOR(212); |
| 240637 | when 213 => BIT_MUX <= PT_VECTOR(213); |
| ed60ee | when 214 => BIT_MUX <= PT_VECTOR(214); |
| e276f0 | when 215 => BIT_MUX <= PT_VECTOR(215); |
| 681d29 | when 216 => BIT_MUX <= PT_VECTOR(216); |
| d7af42 | when 217 => BIT_MUX <= PT_VECTOR(217); |
| 5d497f | when 218 => BIT_MUX <= PT_VECTOR(218); |
| e7fca7 | when 219 => BIT_MUX <= PT_VECTOR(219); |
| 350c9e | when 220 => BIT_MUX <= PT_VECTOR(220); |
| 467d7e | when 221 => BIT_MUX <= PT_VECTOR(221); |
| 03db5e | when 222 => BIT_MUX <= PT_VECTOR(222); |
| 48b087 | when 223 => BIT_MUX <= PT_VECTOR(223); |
| b7ab99 | when 224 => BIT_MUX <= PT_VECTOR(224); |
| aeaf40 | when 225 => BIT_MUX <= PT_VECTOR(225); |
| f27c2b | when 226 => BIT_MUX <= PT_VECTOR(226); |
| 3317f2 | when 227 => BIT_MUX <= PT_VECTOR(227); |
| dba417 | when 228 => BIT_MUX <= PT_VECTOR(228); |
| 9721ce | when 229 => BIT_MUX <= PT_VECTOR(229); |
| 0947cb | when 230 => BIT_MUX <= PT_VECTOR(230); |
| 8ec212 | when 231 => BIT_MUX <= PT_VECTOR(231); |
| cc9079 | when 232 => BIT_MUX <= PT_VECTOR(232); |
| 68fba0 | when 233 => BIT_MUX <= PT_VECTOR(233); |
| d9e0be | when 234 => BIT_MUX <= PT_VECTOR(234); |
| 74b667 | when 235 => BIT_MUX <= PT_VECTOR(235); |
| 7a370c | when 236 => BIT_MUX <= PT_VECTOR(236); |
| 915cd5 | when 237 => BIT_MUX <= PT_VECTOR(237); |
| 900130 | when 238 => BIT_MUX <= PT_VECTOR(238); |
| e16ae9 | when 239 => BIT_MUX <= PT_VECTOR(239); |
| 3abe2f | when 240 => BIT_MUX <= PT_VECTOR(240); |
| 6d5f6 | when 241 => BIT_MUX <= PT_VECTOR(241); |
| 1d999d | when 242 => BIT_MUX <= PT_VECTOR(242); |
| 570244 | when 243 => BIT_MUX <= PT_VECTOR(243); |
when 244 => BIT_MUX <= PT_VECTOR(244);
d87283 when 245 => BIT_MUX <= PT_VECTOR(245);
e2cee8 when 246 => BIT_MUX <= PT_VECTOR(246);
ea531 when 247 => BIT_MUX <= PT_VECTOR(247);
f88d4 when 248 => BIT_MUX <= PT_VECTOR(248);
8c930d when 249 => BIT_MUX <= PT_VECTOR(249);
acf508 when 250 => BIT_MUX <= PT_VECTOR(250);
4a9ed1 when 251 => BIT_MUX <= PT_VECTOR(251);
8b22ba when 252 => BIT_MUX <= PT_VECTOR(252);
c64963 when 253 => BIT_MUX <= PT_VECTOR(253);
6c527d when 254 => BIT_MUX <= PT_VECTOR(254);
74394 when 255 => BIT_MUX <= PT_VECTOR(255);

when others => BIT_MUX <= '0';

end case;

end process DECODER_PR;

---------------------------
end beh;

---------------------------
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity P is
  port(
    P_IN : in std_logic_vector(31 downto 0);
    P_OUT : out std_logic_vector(31 downto 0)
  );
end P;

architecture beh of P is

  subtype small_integer is INTEGER range 0 to 31;
  type P_TYPE is array(0 to 31) of small_integer;

  signal P_TABLE : P_TYPE;

begin

  P_TABLE <=
    (11,17, 5,27,25,10,20, 0,
     13,21, 3,28,29, 7,18,24,
     31,22,12, 6,26, 2,16, 8,
     14,30, 4,19, 1, 9,15,23);

  P_PR: process(P_TABLE,P_IN)
  begin
    for i in 0 to 31 loop
      P_OUT(P_TABLE(i)) <= P_IN(i);
    end loop;
  end process P_PR;

end beh;
Chapter 6: Chip Source Code

6-27

--344c 001e7d76d3803000a Page 1 of pc1.vhd

bb997d ----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---
Chapter 6: Chip Source Code

--fb57 001701367ee800300b Page 1 of pc2.vhd

bb997d ---------------|-------------------|-------------------|-------------------|-------------------|-------------------|-------------------|
025f32 -- Author -------: Tom Vu
D00864 -- Date -----------: 10/02/97
83dd69 -- Description : -- Generate Permutation Choice #2
69c137 -- Function: Array has the table which tells the mapping
64625a --

fb7af library ieee;
4011e9 use IEEE.std_logic_1164.all;
63da83 use IEEE.std_logic_arith.all;
30e105 use IEEE.std_logic_unsigned.all;
33af5a
78af5a
8f625a

cd0fbb entity PC2 is
49af5a
c614be port(...
bbba1f -> KEY_IN :: std_logic_vector(55 downto 0);
9ae70b -> KEY_OUT :: std_logic_vector(47 downto 0);
ca737c ...
60af5a
214683 end PC2;
85af5a
59625a
3197f8 architecture beh of PC2 is
27625a ...
3888c2 subtype small_integer is INTEGER range 0 to 55;
2c5861 type PC2-TYPE is array(0 to 47) of small_integer;
9494fc signal PC2-TABLE : PC2-TYPE;
476af5a
620f89 begin
a8af5a
7b87fd PC2-TABLE <= (24,27,20,6,14,10,3,22,  
bcb9d5 -> 0,17,7,12,8,23,11,5,  
ed8af2a -> 16,26,1,9,19,25,4,15,  
60ed61 -> 54,43,36,29,49,40,48,30,  
f07eaa -> 52,44,37,33,46,35,50,41,  
f9c553 -> 28,53,51,55,32,45,39,42);  
44625a --

9e1c95 Permutation_choice_2: process(KEY_IN,PC2-TABLE)
34625a --
f30899 begin
a8af5a

7b87fd for i in 0 to 47 loop
683e9 -> KEY_OUT(i) <= KEY_IN(PC2-TABLE(i));  
5bb88a end loop;
70625a --
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity REG_RDWR is
  port(
    RST_N: in std_logic;
    BOARD_EN: in std_logic;
    ALE: in std_logic;
    ADDSEL1: in std_logic;
    WRB: in std_logic;
    RDB: in std_logic;
    ADDSEL2: in std_logic;
    AA_IN: in std_logic;
    ABGR: in std_logic_vector(7 downto 0);
    CHIP_ID: in std_logic_vector(7 downto 0);
    SEARCH_OUT: in std_logic_vector(23 downto 0);
    SELECT_ONE: in std_logic_vector(23 downto 0);
    SEARCH_IN: in std_logic_vector(23 downto 0);
    CHIP_EN: out std_logic;
    AA_OUT: out std_logic;
    CHIP_AA_OUT: out std_logic;
    EXTRA_XOR: out std_logic;
    USE_CBC: out std_logic;
    PT_XOR_MASK: out std_logic_vector(63 downto 0);
    PT_BYTE_MASK: out std_logic_vector(7 downto 0);
   chip_AA: out std_logic_vector(255 downto 0);
    C0: out std_logic_vector(63 downto 0);
    C1: out std_logic_vector(63 downto 0);
    DATA1: in std_logic_vector(7 downto 0);
    DATA0: in std_logic_vector(7 downto 0);
  );
end REG_RDWR;
architecture beh of REG_RDWR is
begin
end architecture beh;
--9b90 00080b9fa880300c Page 2 of reg_rgwrd.vhd

98a4177 CHIP_REG <= (others => '0');
8eb4c0 elsif (ALE'event and ALE = '1') then
63a5b5
ff2241 if ((BOARD_EN = '1') and (ADDSEL1 = '1')) then
Da9054 CHIP_REG <= ADDR;
7c62af end if;
2962af end if;
39d83c
10abc end process CHIP_ID_REG_PR;

67a653b facea READ_PR: process (PT_VECTOR_REG, PT_XOR_MASK_REG,
8d07abc => PT_BYTE_MASK_REG, SEARCH_INFO_REG, CIPHER0, CIPHER1,
84d7e79 => SEARCH_IN, SELECT_ONE, ALL_ACTIVE, AA_OUT_BAK,
70735f => CHIP_EN_BAK, ADDSEL2, RDB, ADDR, BAA_EN)
4e5356 begin...

46a6b6 begin (CHIP_EN_BAK = '1') and (ADDSEL2 = '0') and (RDB = '0') then
46bac7 case ADDR is
98a5fa
95f5a7 when "00000000" => DATAO <= PT_VECTOR_REG(0);
505355 when "00000001" => DATAO <= PT_VECTOR_REG(1);
90374e when "00000010" => DATAO <= PT_VECTOR_REG(2);
76669e when "00000011" => DATAO <= PT_VECTOR_REG(3);
9d2036 when "00000100" => DATAO <= PT_VECTOR_REG(4);
6f8099 when "00000101" => DATAO <= PT_VECTOR_REG(5);
7b4893 when "00000110" => DATAO <= PT_VECTOR_REG(6);
93144b when "00000111" => DATAO <= PT_VECTOR_REG(7);
8a167a when "00001000" => DATAO <= PT_VECTOR_REG(8);
7e4a92 when "00001001" => DATAO <= PT_VECTOR_REG(9);
b28a59 when "00001010" => DATAO <= PT_VECTOR_REG(10);
ace73f when "00001011" => DATAO <= PT_VECTOR_REG(11);
921259 when "00001100" => DATAO <= PT_VECTOR_REG(12);
185f13 when "00001101" => DATAO <= PT_VECTOR_REG(13);
3e54fa when "00001110" => DATAO <= PT_VECTOR_REG(14);
1c199c when "00001111" => DATAO <= PT_VECTOR_REG(15);
2dcc6d when "00010000" => DATAO <= PT_VECTOR_REG(16);
67d011 when "00010001" => DATAO <= PT_VECTOR_REG(17);
2df716 when "00010010" => DATAO <= PT_VECTOR_REG(18);
5c3270 when "00010011" => DATAO <= PT_VECTOR_REG(19);
9f2674 when "00010100" => DATAO <= PT_VECTOR_REG(20);
f0e518 when "00010101" => DATAO <= PT_VECTOR_REG(21);
1e123b when "00011000" => DATAO <= PT_VECTOR_REG(22);
98c05d when "00011011" => DATAO <= PT_VECTOR_REG(23);
90428e when "00100000" => DATAO <= PT_VECTOR_REG(24);
a4518 when "00011010" => DATAO <= PT_VECTOR_REG(25);
a876ca when "00110100" => DATAO <= PT_VECTOR_REG(26);
763ba1 when "00110101" => DATAO <= PT_VECTOR_REG(27);
9379f3 when "00110110" => DATAO <= PT_VECTOR_REG(28);
7f3495 when "00110111" => DATAO <= PT_VECTOR_REG(29);
0c9a56 when "00111000" => DATAO <= PT_VECTOR_REG(30);
94d730 when "00111011" => DATAO <= PT_VECTOR_REG(31);
64a5f5 when "00100000" => DATAO <= PT_XOR_MASK_REG(0);
5cd063 when "00100001" => DATAO <= PT_XOR_MASK_REG(1);
78f89a when "00100010" => DATAO <= PT_XOR_MASK_REG(2);
a9e918 when "00100011" => DATAO <= PT_XOR_MASK_REG(3);
4ae518 when "00111000" => DATAO <= PT_XOR_MASK_REG(4);
6bb492 when "00111010" => DATAO <= PT_XOR_MASK_REG(5);
9f3829 when "00100110" => DATAO <= PT_XOR_MASK_REG(6);
ae21ab when "00101111" => DATAO <= PT_XOR_MASK_REG(7);
8af5a when "00101000" => DATAO <= CIPHER0(0);
e552d5 when "00110000" => DATAO <= CIPHER1(0);
-- 4711 000c5caea668003000c Page 3 of reg_rdwr.vhd

95fda when "00110001" => DATAO <= CIPHER(1);
e64451 when "00110010" => DATAO <= CIPHER(2);
dd0a2 when "00110011" => DATAO <= CIPHER(3);
75384 when "00110100" => DATAO <= CIPHER(4);
a17617 when "00110101" => DATAO <= CIPHER(5);
18c8c when "00110110" => DATAO <= CIPHER(6);
a287f when "00110111" => DATAO <= CIPHER(7);
b74798 when "00111000" => DATAO <= PT_BYTE_MASK_REG;
1ba7e9 when "00111111" => DATAO <= "000" & BAA_EN &
d268c2 : AA_OUT_BAK & ALL_ACTIVE & SEARCH_INFO_REG(1 downto 0)

0f548f when "01000111" => DATAO <= "00000000" & SELECT_ONE(0) & SEARCH_IN(0);
65e9de when "01001111" => DATAO <= "00000000" & SELECT_ONE(1) & SEARCH_IN(1);
b0a4d4 when "01010111" => DATAO <= "00000000" & SELECT_ONE(2) & SEARCH_IN(2);
461085 when "01011111" => DATAO <= "00000000" & SELECT_ONE(3) & SEARCH_IN(3);
2dd7fe when "01100111" => DATAO <= "00000000" & SELECT_ONE(4) & SEARCH_IN(4);
3c63af when "01101111" => DATAO <= "00000000" & SELECT_ONE(5) & SEARCH_IN(5);
e327a5 when "01110111" => DATAO <= "00000000" & SELECT_ONE(6) & SEARCH_IN(6);
4493f4 when "01111111" => DATAO <= "00000000" & SELECT_ONE(7) & SEARCH_IN(7);
9d13d9 when "10000111" => DATAO <= "00000000" & SELECT_ONE(8) & SEARCH_IN(8);
22a788 when "10001111" => DATAO <= "00000000" & SELECT_ONE(9) & SEARCH_IN(9);
942170 when "10101111" => DATAO <= "00000000" & SELECT_ONE(10) & SEARCH_IN(10);
a893d8 when "10110111" => DATAO <= "00000000" & SELECT_ONE(11) & SEARCH_IN(11);
64a860 when "10111111" => DATAO <= "00000000" & SELECT_ONE(12) & SEARCH_IN(12);
1a94a when "11000111" => DATAO <= "00000000" & SELECT_ONE(13) & SEARCH_IN(13);
a9a4da when "11010111" => DATAO <= "00000000" & SELECT_ONE(14) & SEARCH_IN(14);
3718a7 when "11100111" => DATAO <= "00000000" & SELECT_ONE(15) & SEARCH_IN(15);
35d7bf when "11000011" => DATAO <= "00000000" & SELECT_ONE(16) & SEARCH_IN(16);
46b6c2 when "11001111" => DATAO <= "00000000" & SELECT_ONE(17) & SEARCH_IN(17);
52f4fb when "11011111" => DATAO <= "00000000" & SELECT_ONE(18) & SEARCH_IN(18);
b64876 when "11011011" => DATAO <= "00000000" & SELECT_ONE(19) & SEARCH_IN(19);
3899a6 when "11100111" => DATAO <= "00000000" & SELECT_ONE(20) & SEARCH_IN(20);
8525db when "11101111" => DATAO <= "00000000" & SELECT_ONE(21) & SEARCH_IN(21); 
c2829b when "11110111" => DATAO <= "00000000" & SELECT_ONE(22) & SEARCH_IN(22);
023ee6 when "11111111" => DATAO <= "00000000" & SELECT_ONE(23) & SEARCH_IN(23);
776aab when others => DATAO <= (others => 'Z');

d492b5 end case;
56523c else
a35d6b DATAO <= (others => 'Z');
6b2c2b end process READ_PR;
4d5356
f9ab46 PT_VECTOR_PR: process(RST_\_N, WRB)
795356

9f3dcd if (RST_\_N = '0') then
9bd72a for i in 0 to 31 loop
1953d94 PT_\_VECTOR_\_REG(i) <= (others => '0');
4b7aa9 end loop;
1faf5a
a8d7fd for i in 0 to 7 loop
bd55c9 PT_\_XOR_\_MASK_\_REG(i) <= (others => '0');
5fd5f4 CIPHER0(i) <= (others => '0');
6adb2f CIPHER1(i) <= (others => '0');
a37aa9 end loop;
3fa5a
40889f PT_\_BYTE_\_MASK_\_REG <= (others => '0');
f15de2 SEARCH_INFO_\_REG <= (others => '0');
f7af5a
f9a9e0 else (WRB'event and WRB = '1') then
6af5a
32e1c4 if ((CHIP\_EN\_BAK = '1') and (ADDSEL2 = '0')) then
48bca7 case ADDR is
51ed1 when "00000000" => PT_\_VECTOR_\_REG(0) <= DATAI;
94a1f1 when "00000001" => PT_\_VECTOR_\_REG(1) <= DATAI;
69b156 when "00000010" => PT_\_VECTOR_\_REG(2) <= DATAI;
86f498 when "00000011" => PT_\_VECTOR_\_REG(3) <= DATAI;
cef85c when "00000100" => PT_\_VECTOR_\_REG(4) <= DATAI;
82b592 when "00000101" => PT_\_VECTOR_\_REG(5) <= DATAI;
c5a9db when "00000110" => PT_\_VECTOR_\_REG(6) <= DATAI;
018b2c when "00000111" => PT_\_VECTOR_\_REG(7) <= DATAI;
end case;
endif;
-- f423 00420F0478083000c Page 4 of reg_rdwrd.vhd

47e155 when "00000111" => PTVECTOR_REG(7) <= DATAI;
d6afa6 when "00001000" => PTVECTOR_REG(8) <= DATAI;
142f38 when "00001001" => PTVECTOR_REG(9) <= DATAI;
409fd3 when "00001010" => PTVECTOR_REG(10) <= DATAI;
a5cb3 when "00001011" => PTVECTOR_REG(11) <= DATAI;
b035f when "00001100" => PTVECTOR_REG(12) <= DATAI;
d3618f when "00001101" => PTVECTOR_REG(13) <= DATAI;
512528 when "00001110" => PTVECTOR_REG(14) <= DATAI;
8e7158 when "00001111" => PTVECTOR_REG(15) <= DATAI;
e39f93 when "00010000" => PTVECTOR_REG(16) <= DATAI;
a1cbe3 when "00010001" => PTVECTOR_REG(17) <= DATAI;
1fa22c when "00010010" => PTVECTOR_REG(18) <= DATAI;
4af65c when "00010011" => PTVECTOR_REG(19) <= DATAI;
150dc when "00010100" => PTVECTOR_REG(20) <= DATAI;
15f97c when "00010101" => PTVECTOR_REG(21) <= DATAI;
cb06 when "00010110" => PTVECTOR_REG(22) <= DATAI;
a9f1f when "00010111" => PTVECTOR_REG(23) <= DATAI;
17ada8 when "00011000" => PTVECTOR_REG(24) <= DATAI;
28f9d8 when "00011001" => PTVECTOR_REG(25) <= DATAI;
a7abc when "00011010" => PTVECTOR_REG(26) <= DATAI;
17f1bb when "00011011" => PTVECTOR_REG(27) <= DATAI;
803a3b when "00011100" => PTVECTOR_REG(28) <= DATAI;
916e4b when "00011101" => PTVECTOR_REG(29) <= DATAI;
fc60 when "00011110" => PTVECTOR_REG(30) <= DATAI;
7d900 when "00011111" => PTVECTOR_REG(31) <= DATAI;
ecaf5a when "00000000" => PTXORMASK_REG(0) <= DATAI;
b621be when "00000001" => PTXORMASK_REG(1) <= DATAI;
a0b3 when "00000010" => PTXORMASK_REG(2) <= DATAI;
8dc6b when "00000011" => PTXORMASK_REG(3) <= DATAI;
d1c5c1 when "00000100" => PTXORMASK_REG(4) <= DATAI;
b22c55 when "00000101" => PTXORMASK_REG(5) <= DATAI;
4262d4 when "00000110" => PTXORMASK_REG(6) <= DATAI;
116a40 when "00000111" => PTXORMASK_REG(7) <= DATAI;
4aaf5a when "00001000" => CIPHER0(0) <= DATAI;
abe2f when "00001001" => CIPHER0(1) <= DATAI;
ed70e when "00001010" => CIPHER0(2) <= DATAI;
2827eb when "00001011" => CIPHER0(3) <= DATAI;
f65cf when "00001100" => CIPHER0(4) <= DATAI;
180d2 when "00001101" => CIPHER0(5) <= DATAI;
37f9 when "00001110" => CIPHER0(6) <= DATAI;
9ac8ee when "00001111" => CIPHER0(7) <= DATAI;
69af5a

bd187 when "00010000" => CIPHER1(0) <= DATAI;
ea4f2 when "00010001" => CIPHER1(1) <= DATAI;
58dde3 when "00010010" => CIPHER1(2) <= DATAI;
f4a8b6 when "00010011" => CIPHER1(3) <= DATAI;
70f7c8 when "00010100" => CIPHER1(4) <= DATAI;
86a0 when "00010101" => CIPHER1(5) <= DATAI;
ef3266 when "00010110" => CIPHER1(6) <= DATAI;
f568 when "00010111" => CIPHER1(7) <= DATAI;
1daf5a

720f9 when "00011000" => BYTEMASK_REG <= DATAI;
16af5a

1a1063 when "00011111" => SEARCHINFO_REG <= DATAI;
70af5a

7aaf5a

27af5a

234c2p end if;
3a8c29 when others => null;
a58259 end case;
d262af end if;
cf62af end if;
3ed83c

cf926a end process PTVECTOR_PR;
0cd83c

5a63e0 PTVECTOR <= PTVECTOR_REG(31) & PTVECTOR_REG(30) & PTVECTOR_REG(29) & PTVECTOR_REG(28) & PTVECTOR_REG(27) & PTVECTOR_REG(26) & PTVECTOR_REG(25) & PTVECTOR_REG(24) & PTVECTOR_REG(23) & PTVECTOR_REG(22) & PTVECTOR_REG(21) & PTVECTOR_REG(20) &
--88e5 001f556d880300c Page 5 of reg_rdr.vhd

048028 b PT_VECTOR_REG(19) & PT_VECTOR_REG(18) & PT_VECTOR_REG(17) & PT-
216203 VECTOR_REG(16) &
216203 VECTOR_REG(15) & PT_VECTOR_REG(14) & PT_VECTOR_REG(13) & PT-
474f13 VECTOR_REG(12) &
d33ced PT_VECTOR_REG(11) & PT_VECTOR_REG(10) & PT_VECTOR_REG(9) & PT-
400ee7 VECTOR_REG(8) &
400ee7 VECTOR_REG(7) & PT_VECTOR_REG(6) & PT_VECTOR_REG(5) & PT-
d19f75 &
867f9d PT_VECTOR_REG(4) &
fadb33 PT_VECTOR_REG(3) & PT_VECTOR_REG(2) & PT_VECTOR_REG(1) & PT-
ab7f1e VECTOR_REG(0);
9daf5a f79fa6 PT_VECTOR_REG(7) & PT_VECTOR_REG(6) & PT_VECTOR_REG(5) &
f47e6 PT_VECTOR_REG(4) &
9332e5 PT_VECTOR_REG(3) & PT_VECTOR_REG(2) & PT_VECTOR_REG(1) &
708f30 PT_VECTOR_REG(0);
c6af5a bce0a8 C1 <= CIPHER1(7) & CIPHER1(6) & CIPHER1(5) & CIPHER1(4) &-
1638e0 CIPHER1(3) & CIPHER1(2) & CIPHER1(1) & CIPHER1(0);
1d1a2d C0 <= CIPHER0(7) & CIPHER0(6) & CIPHER0(5) & CIPHER0(4) &-
71bfc6 CIPHER0(3) & CIPHER0(2) & CIPHER0(1) & CIPHER0(0);
58af5a b05356 --
2ed85c -
585536 5496b8b PT_BYTE_MASK <= PT_BYTE_MASK_REG;
38bb8c9 USE_CBC <= SEARCH_INFO_REG(0);
1a064 EXTRAXOR <= SEARCH_INFO_REG(1);
12b4d8 BAA_EN <= SEARCH_INFO_REG(4);
3073ee AA_OUT_BAK <= AA_IN and ALL_ACTIVE when (BAA_EN = '1') else AA_IN;
5d2b0d AA_OUT_BAK <= AA_OUT_BAK;
9aaf5a b0351c ALL_ACTIVE <= (SEARCH_OUT(23) and SEARCH_OUT(22) and SEARCH_OUT(21) and S-
1e4820 SEARCH_OUT(19) and SEARCH_OUT(18) and SEARCH_OUT(17) and S-
2e0b5f SEARCH_OUT(16) and
2e0b5f SEARCH_OUT(15) and SEARCH_OUT(14) and SEARCH_OUT(13) and S-
4291e9 SEARCH_OUT(12) and
4291e9 SEARCH_OUT(11) and SEARCH_OUT(10) and SEARCH_OUT(9) &
68357c SEARCH_OUT(8) and
68357c SEARCH_OUT(7) &
8650b9 SEARCH_OUT(6) &
8650b9 SEARCH_OUT(5) &
6b8753 SEARCH_OUT(3) &
6b8753 SEARCH_OUT(2) &
8b76b6 SEARCH_OUT(0);
7a0f7f CHIP_AA_OUT <= ALL_ACTIVE;
8807f3 CHIP_EN <= CHIP_EN_BAK;
c45356 --
595536 end beh;
5afa5a
Chapter 6: Chip Source Code

6-34

-- 068f 000b046c24980003000d Page 1 of s_table.vhd

bb997d - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
d8be19 -- Author: Tom Vu
4a0864 -- Date: 10/02/97
dc5b6c -- Description: Create table for lookup values of S function
152717 -- Function: 6 inputs are used to lookup in the table and produce 4 outputs. There are a total of 8 tables.
829e89 --

9b625a --

3e7af library ieee;
2511e9 use IEEE.std_logic_1164.all;
95da83 use IEEE.std_logic_arith.all;
b1e105 use IEEE.std_logic_unsigned.all;
ea6414 use ieee.std_logic_arith.conv_std_logic_vector;

749fa5 entity S_TABLE is
82c026 port - KEY - S-OUT end S_TABLE;

72a250 architecture beh of S_TABLE is
2a625a --

60fa2e subtype small_integer is INTEGER range 0 to 15;
ceed28 type TABLE_TYPE is array(0 to 63) of small_integer;

35a5fa a5occ3 signal S1 : TABLE_TYPE;
6d7f72 signal S2 : TABLE_TYPE;
4e49e2 signal S3 : TABLE_TYPE;
ebfba1 signal S4 : TABLE_TYPE;
4275b1 signal S5 : TABLE_TYPE;
57e9f0 signal S6 : TABLE_TYPE;
98e180 signal S7 : TABLE_TYPE;
5a85b7 signal S8 : TABLE_TYPE;

74625a function lookup(signal table : in TABLE_TYPE;
657295 signal key : in std_logic_vector(5 downto 0))
87f5fa return std_logic_vector is
5e43f3 b0af5a --

ek77f7 variable row : std_logic_vector(3 downto 0);
c66911 variable col : std_logic_vector(1 downto 0);
0f40ec variable addr : std_logic_vector(5 downto 0);
324c51 variable index : integer;
6166ad variable result : std_logic_vector(3 downto 0);

3f089f begin
fda5fa ddf25 col := key(5) & key(0);
a652a row := key(4 downto 1);
a1f72c addr := col & row;
ae23b index := CONV_INTEGER(key);
4af5a 87f5ba result := CONV_STD_LOGIC_VECTOR(table(index),4);
39af5a 7214bd return result;
eka5a 9b0c01 end lookup;

36625a --

330189 begin
38625a --

0671a2 S1 <= (13, 1, 2, 15, 8, 13, 4, 8, 6, 10, 15, 3, 11, 7, 1, 4, 

d8e826) <= (10, 12, 9, 3, 3, 6, 14, 11, 5, 0, 0, 14, 12, 9, 7, 2, 

eaf5a) <= (7, 2, 11, 1, 6, 14, 1, 7, 9, 4, 12, 10, 14, 8, 2, 13, 

106ad <= (0, 15, 6, 12, 10, 9, 13, 0, 15, 3, 3, 5, 5, 6, 8, 11);

dba5a baf37 S2 <= (4, 13, 11, 0, 2, 11, 14, 7, 15, 4, 0, 9, 8, 1, 13, 10,
Chapter 6: Chip Source Code

6-35

-- 1b4d 00142225b498003000 d Page 2 of s_table.vhd

6aef27 <= 63, 14, 12, 3, 9, 5, 7, 12, 5, 2, 10, 15, 6, 8, 1, 6,
dfeb0f <= 1, 6, 4, 11, 11, 13, 13, 8, 12, 1, 3, 4, 7, 10, 14, 7,
088e69 <= 10, 9, 15, 5, 6, 0, 8, 15, 0, 14, 5, 2, 9, 3, 2, 12);
d2af5a <= 4ad185 S3 <= (12, 10, 1, 15, 10, 4, 15, 2, 9, 7, 2, 12, 6, 9, 8, 5,
b53629 <= 0, 6, 13, 1, 3, 13, 4, 14, 14, 0, 7, 11, 5, 3, 11, 8,
de712b <= 9, 4, 14, 3, 15, 2, 5, 12, 2, 9, 8, 5, 12, 15, 3, 10,
37d78f <= 7, 11, 0, 14, 4, 1, 10, 7, 1, 6, 13, 0, 11, 8, 6, 13);
d2af5a <= c63d3c S4 <= (2, 14, 12, 11, 4, 2, 1, 12, 7, 4, 10, 7, 11, 13, 6, 1, 
d80ade <= 8, 5, 5, 0, 3, 15, 15, 10, 13, 3, 0, 9, 14, 8, 9, 6,
e10d88 <= 4, 11, 2, 8, 1, 12, 11, 7, 10, 1, 13, 14, 7, 2, 8, 13,
d84f6fa <= 15, 6, 9, 2, 12, 8, 5, 9, 6, 10, 3, 4, 0, 5, 14, 3);
aeaf5a <= b63631 S5 <= (7, 13, 13, 8, 14, 11, 3, 5, 0, 6, 6, 15, 9, 0, 10, 3,
f08223 <= 1, 4, 2, 7, 8, 2, 5, 12, 11, 1, 12, 10, 4, 14, 15, 9,
a72f41 <= 10, 3, 6, 15, 9, 0, 0, 6, 12, 10, 11, 1, 7, 13, 13, 8,
h492e5 <= 15, 9, 1, 4, 3, 5, 14, 11, 5, 12, 2, 7, 8, 2, 4, 14);
30af5a <= e251c8 S6 <= (10, 13, 0, 7, 9, 0, 14, 9, 6, 3, 3, 4, 15, 6, 5, 10,
d88b14 <= 1, 2, 13, 8, 12, 5, 7, 14, 11, 12, 4, 11, 2, 15, 8, 1,
faa01f <= 13, 1, 6, 10, 4, 13, 9, 0, 8, 6, 15, 9, 3, 8, 0, 7,
d2f2c2 <= 11, 4, 1, 15, 2, 14, 12, 3, 5, 11, 10, 5, 14, 2, 7, 12);
49a5a <= bd692c S7 <= (15, 3, 1, 13, 8, 4, 14, 7, 6, 15, 11, 2, 3, 8, 4, 14,
b6f95b <= 9, 12, 7, 0, 2, 1, 13, 10, 12, 6, 0, 9, 5, 11, 10, 5,
8c0ccc <= 0, 13, 14, 3, 7, 10, 11, 1, 10, 3, 4, 15, 13, 4, 1, 2,
fa311b <= 5, 11, 8, 6, 12, 7, 6, 12, 9, 0, 3, 5, 2, 14, 15, 9);
d6af5a <= bbb0a S8 <= (14, 0, 4, 15, 13, 7, 1, 4, 2, 14, 15, 2, 11, 13, 8, 1,
2ac15d <= 3, 10, 10, 6, 6, 12, 12, 11, 5, 9, 9, 5, 0, 3, 7, 8,
54f1e <= 4, 15, 1, 12, 14, 8, 8, 2, 13, 4, 6, 9, 2, 1, 11, 7,
834ce4 <= 15, 5, 12, 11, 9, 3, 7, 14, 13, 10, 0, 5, 6, 0, 13);
25af5a <= bd8694 S_OUT <= Look up (S8, KEY (47 downto 42)) &
535b55 <= Look up (S7, KEY (41 downto 36)) &
54f229 <= Look up (S6, KEY (35 downto 30)) &
dd873b <= Look up (S5, KEY (29 downto 24)) &
acd315 <= Look up (S4, KEY (23 downto 18)) &
9d4724 <= Look up (S3, KEY (17 downto 12)) &
4af3c2 <= Look up (S2, KEY (11 downto 6)) &
b5a317 <= Look up (S1, KEY (5 downto 0)) ;
daaf5a --
95d25a --
bcdb08a end beh;
k2625a --
Chapter 6: Chip Source Code

-- 4fac 00050e451e18003000 Page 1 of search.vhd

bb997d ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |
aa533a -- Author --------: Tom Vu ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |
3a917e -- Date ----------: 09/07/97 ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |
857268 -- Description ->: Search Unit ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |
2d5356 ------------------- ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |

057af library ieee;
d811e9 use IEEE.std_logic_1164.all;
e3da83 use IEEE.std_logic_arith.all;
a0e105 use IEEE.std_logic_unsigned.all;
5b5356 ------------------- ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |

53cbd6 entity SEARCH_UNIT is
72af5a 8dec7a port( CLK : in std_logic;
ffe277 RST_N : in std_logic;
3d6737 WB : in std_logic;
77976b RDB : in std_logic;
5bbaa SEARCH : in std_logic;
65a2a9 EXTRA_XOR : in std_logic;
39dbfc USE_CBC : in std_logic;
d2913 ADDR_KEY : in std_logic_vector(6 downto 0);
28af5a 41fdcc DATAI : in std_logic_vector(7 downto 0);
39e965 PT_BYTE_MASK : in std_logic_vector(7 downto 0);
07b25c PT_XOR_MASK : in std_logic_vector(63 downto 0);
00b06f PT_VEC : in std_logic_vector(255 downto 0);
8f8d2e C0 : in std_logic_vector(63 downto 0);
2c37cb C1 : in std_logic_vector(63 downto 0);
ab6756 KEY_OUT : out std_logic_vector(55 downto 0);
ae7474 DES_OUTPUT : out std_logic_vector(63 downto 0);
47f098 MATCH_OUT : out std_logic;
26f4f SELECT_ONE : out std_logic;
ce450 SEARCH_OUT : out std_logic;
60b1c CLEAR_SEARCH: out std_logic;
45a552 DATAO : out std_logic_vector(7 downto 0);
f2737c );
31af5a
9a5f5a
9b3e22 end SEARCH_UNIT;
13af5a
195356 ------------------- ------------------- |------------------- |------------------- |------------------- |------------------- |------------------- |

8cacc3e architecture beh of SEARCH_UNIT is
fc5356 8f0e4c type DATA8_ARRAY is array(7 downto 0) of std_logic_vector(7 downto 0);
50af5a

65a698 signal MESSAGE : std_logic_vector(63 downto 0);
63ea9a signal IP_KEY : std_logic_vector(63 downto 0);
1d2c74 signal DES_OUT : std_logic_vector(63 downto 0);
d77887 signal EXTRA_XOR_OUT : std_logic_vector(63 downto 0);
e1599 signal SHIFT_REG : DATA8_ARRAY;
227ca3 signal KEY : std_logic_vector(55 downto 0);
fb8eac signal D_KEY : std_logic_vector(31 downto 0);
d9e79c signal MESG_LEFT : std_logic_vector(31 downto 0);
60b1c signal CNT : std_logic_vector(4 downto 0);
6b7f7a signal BIT_SHIFT_REG : std_logic_vector(7 downto 0);
e1b6d1 signal TEMP_VECTOR : std_logic_vector(3 downto 0);
c07eb2 signal WRB : std_logic;
7a44a8 signal WR_STROBE : std_logic;
d2093 signal DONE : std_logic;
0e18c7 signal STARTDES : std_logic;
ae027a signal MATCH : std_logic;
ecaec6 signal MATCH_DLY_CYCLE1 : std_logic;
4676b8 signal MATCH_DLY_CYCLE2 : std_logic;
73eb38 signal FALSE_MATCH : std_logic;
bc3de8 signal SEARCH_DLY1 : std_logic;
7c5468 signal SEARCH_DLY2 : std_logic;
d48b55 signal SEARCH_DLY3 : std_logic;
7b8fe9 signal SEARCHING : std_logic;
0d1d38 signal SEARCHING_DLY : std_logic;
1c6abc signal LOAD : std_logic;
6dc677 signal FIRST_TIME1 : std_logic;
e0af9c signal FIRST_TIME2 : std_logic;
50cc54 signal FIRST_LOAD : std_logic;
--ac0f 0005aea51108000300e Page 2 of search.vhd
720e91 signal SELECT1 : std_logic;
c99f27 signal SELECT1_DLY : std_logic;
339514 signal KEY_ODD_DLY1 : std_logic_vector(1 downto 0);
d4a2e0 signal KEY_ODD_DLY2 : std_logic_vector(1 downto 0);
98ae92 signal CHECKSAME_KEY : std_logic;
ec29a4 signal KEY Incre : std_logic;
0c0f34 signal KEY_DECKn : std_logic;
fa2d2c signal PRE_DONE : std_logic;
0beef1 signal CNT_EQ_1 : std_logic;
78323d signal CNT_GT_10 : std_logic;
b3d57b signal CNT_LE_10 : std_logic;
571491 signal CNT_EQ_10 : std_logic;
6f4a5e signal FIRST_DESn : std_logic;
91af1e signal RESET_SEARCHING : std_logic;
f70f75 signal CLEAR_SEARCH_BAK : std_logic;
d1af5a
6f18d5 signal EXTRA_SELECT : std_logic_vector(2 downto 0);
ebaf5a
bebef5 signal BIT_MUX : std_logic;
23af5a
326af1 component DES
baec7a port (CLK, .....
792e77 : std_logic;
61d3de : std_logic;
370f57 : std_logic_vector(63 downto 0);
2bd44f : std_logic_vector(55 downto 0);
919930 : std_logic;
efffe96 : std_logic_vector(4 downto 0);
h93850 : std_logic_vector(63 downto 0);
3c737c .....
cc2e0c end component;
77af5a
bba5f9 component MUX256
3b14be port ( ..
ec5af9 .....
6cc584 : std_logic_vector(7 downto 0);
61809f : std_logic_vector(255 downto 0);
d9af4d : std_logic;
5e737c .....
13e2e0c end component;
ffaf5a
270f89 begin
d120f2 M256: MUX256
d42a10 port map ( ..
9a87e8 : std_logic;
78715f : std_logic;
96737c .....
a7af5a
c90b40 DES1: DES
9bc589 port map ( ..
4c2b3b : std_logic;
6ee5ac : std_logic;
87bbda : std_logic;
5e184c : std_logic;
4e5205 : std_logic;
acfb8c : std_logic;
cb737c .....
4e4320 MESSAGE <= C0 when (SELECT1 = '0') else C1;
5b5356 .....
1ec7f50 PCSETSEARCH_PR: process(RST_N,CLK)
8f5356 .....
8a9ebc begin ..
9d6118 if RST_N = '0' then
d18115 .....
0f6e7c .....
ce380c .....
4e4c65 .....
55684g .....
add7d for i in 0 to 7 loop
5a3d5f .....

Chapter 6: Chip Source Code

--9c5f 00039b83b5803000e Page 4 of search.vhd

7a7e5e \begin{verbatim}
... (BIT_SHIFT_REG(2) = '1' or (PT_BYTE_MASK(2) = '1')) and
003373 \begin{verbatim}
... (BIT_SHIFT_REG(3) = '1' or (PT_BYTE_MASK(3) = '1')) and
ac0a72 \begin{verbatim}
... (BIT_SHIFT_REG(4) = '1' or (PT_BYTE_MASK(4) = '1')) and
fd0c8f \begin{verbatim}
... (BIT_SHIFT_REG(5) = '1' or (PT_BYTE_MASK(5) = '1')) and
080af9 \begin{verbatim}
... (BIT_SHIFT_REG(6) = '1' or (PT_BYTE_MASK(6) = '1')) and
49dbf9 \begin{verbatim}
... (BIT_SHIFT_REG(7) = '1' or (PT_BYTE_MASK(7) = '1')) then
44af5a \begin{verbatim}
94087d \begin{verbatim}
... MATCH <= '1';
9db081 \begin{verbatim}
\begin{verbatim}
end if;
289f1c \begin{verbatim}
\begin{verbatim}
if (FIRST_LOAD = '1') then
2e1466 \begin{verbatim}
... MATCH <= '0';
a457d0 \begin{verbatim}
\end if;
029f1c \begin{verbatim}
\begin{verbatim}
\end if;
3466bf \begin{verbatim}
... MATCH_DLY_CYCLE2 <= MATCH_DLY_CYCLE1;
814dc2 \begin{verbatim}
... MATCH_DLY_CYCLE1 <= MATCH;
81730d \begin{verbatim}
\end if;
d39cfc \begin{verbatim}
\begin{verbatim}
9c2e23 \begin{verbatim}
... if (PRE_DONE = '1') then
b6b05a \begin{verbatim}
... KEY_ODD_DLY2 <= KEY_ODD_DLY1;
dd4a58 \begin{verbatim}
... KEY_ODD_DLY1 <= KEY(1 downto 0);
9c57bd \begin{verbatim}
\end if;
79df3f \begin{verbatim}
\end if;
cb400f \begin{verbatim}
\end process MATCH_PR;
e6af5a \begin{verbatim}
\end if;
2e5356 \begin{verbatim}
\end process WRITE_STROBE_PR;
4c53c \begin{verbatim}
WRITE_STROBE_PR: process(RST_N,CLK)
\end if;
645356 \begin{verbatim}
890f89 \begin{verbatim}
begin
6f118 \begin{verbatim}
if RST_N = '0' then
a4b9ef \begin{verbatim}
... WR1B <= '1';
e68ae4 \begin{verbatim}
... WR_STROBEB <= '1';
e884bd \begin{verbatim}
elsif CLK'event and CLK = '1' then
ad4c03 \begin{verbatim}
... WR_STROBEB <= WR1B;
362cfc \begin{verbatim}
... WR1B <= WRB;
2bdf3b \begin{verbatim}
end if;
74a4db \begin{verbatim}
\end process WRITE_STROBE_PR;
4da5a \begin{verbatim}
\end process MATCH_PR;
c33157 \begin{verbatim}
KEY_PR: process(RST_N,CLK)
\end if;
775356 \begin{verbatim}
930f89 \begin{verbatim}
begin
391ca4 \begin{verbatim}
if (RST_N = '0') then
568baf \begin{verbatim}
... KEY <= (others => '0');
684bd \begin{verbatim}
elsif CLK'event and CLK = '1' then
56a63 \begin{verbatim}
... if (WR1B = '0' and ADDR_KEY(0) = '1') then
95911 \begin{verbatim}
... KEY(7 downto 0) <= DATAI;
baca4b \begin{verbatim}
elsif (PRE_DONE = '1') then
ca4a28 \begin{verbatim}
... KEY(7 downto 0) <= D_KEY(7 downto 0);
a3b985 \begin{verbatim}
\end if;
c368d3 \begin{verbatim}
\end if;
fb2249 \begin{verbatim}
... if (WR1B = '0' and ADDR_KEY(1) = '1') then
af95f6 \begin{verbatim}
... KEY(15 downto 8) <= DATAI;
b2ca4b \begin{verbatim}
elsif (PRE_DONE = '1') then
5718d4 \begin{verbatim}
... KEY(15 downto 8) <= D_KEY(15 downto 8);
3fb985 \begin{verbatim}
\end if;
3868d3 \begin{verbatim}
\end if;
0efa37 \begin{verbatim}
... if (WR1B = '0' and ADDR_KEY(2) = '1') then
8955f1 \begin{verbatim}
... KEY(23 downto 16) <= DATAI;
24ca4b \begin{verbatim}
elsif (PRE_DONE = '1') then
e55971 \begin{verbatim}
... KEY(23 downto 16) <= D_KEY(23 downto 16);
49b985 \begin{verbatim}
\end if;
6d86d3 \begin{verbatim}
\end if;
76b21d \begin{verbatim}
... if (WR1B = '0' and ADDR_KEY(3) = '1') then
b284ca \begin{verbatim}
... KEY(31 downto 24) <= DATAI;
e4ca4b \begin{verbatim}
elsif (PRE_DONE = '1') then
a55581 \begin{verbatim}
... KEY(31 downto 24) <= D_KEY(31 downto 24);
39b985 \begin{verbatim}
\end if;
\end if;
\end if;
Chapter 6: Chip Source Code

--a17f 00076022f608003000e Page 5 of search.vhd

2068d3 ------
e442da ------ if (WR1B = '0' and ADDR_KEY(4) = '1') then
072921 ------ KEY(39 downto 32) <= DATAI;
59b985 ------ end if;
ba8d3 ------ if (WR1B = '0' and ADDR_KEY(5) = '1') then
798b8 ------ KEY(47 downto 40) <= DATAI;
4db985 ------ end if;
9868d3 ------ if (WR1B = '0' and ADDR_KEY(6) = '1') then
5fd28e ------ KEY(55 downto 48) <= DATAI;
ba68d3 ------ end if;
9868d3 ------ if (WR1B = '0' and ADDR_KEY(7) = '1') then
4db985 ------ end if;
1faf5a 1baf5a
f6af5a 26df0b end if;
7b7b3ce end process KEY_PR;
968b86 o DATAO <= (others => 'Z');
1bdf0b end if;
c5a5d7 end process READ_KEY_PR;
8caf5a 36af5a 925356
6ada3c KEY_ALU_PR: process(KEY_DEC,KEY_INCR,KEY)
ad5356 ------
8c0f89 begin
f8af5a 7a5af5a
95b33c end process KEY_ALU_PR;
e8af5a 145356
1b4a7a EXTRA_XOR_PR: process(PT_XOR_MASK,EXTRA_SELECT, DES_OUT, C0)
0e5356 ------
68bf89 begin
dd6898 case EXTRA_SELECT is
1e78af when "00" =>
588585d ------ EXTRA_XOR_OUT <= DES_OUT xor PT_XOR_MASK;
17e77a when "01" =>
76585d ------ EXTRA_XOR_OUT <= DES_OUT xor PT_XOR_MASK;
--de52 000e587fcfc8003000e Page 6 of search.vhd

1f ef90 when "010" =>
   db7fde  "•" EXTRA_XOR_OUT <= ((DES_OUT(63 downto 56) xor DES_OUT(31 downto 24)) &
   ddf1f1d  "•"  ((DES_OUT(55 downto 48) xor DES_OUT(23 downto 16)) &
   656d  "•"  ((DES_OUT(47 downto 40) xor DES_OUT(15 downto 8)) &
   a0d1e7  "•"  ((DES_OUT(39 downto 32) xor DES_OUT(7 downto 0)) &
   bd7639  "•"  ((DES_OUT(31 downto 0)) xor PT_XOR_MASK);
   fbc5e4  "•"

e4aa87 when "101" =>
   b8c89f  "•" EXTRA_XOR_OUT <= DES_OUT xor C0;

71b4ed when "110" =>
   561724  "•" EXTRA_XOR_OUT <= ((DES_OUT(63 downto 56) xor DES_OUT(31 downto 24)) 
   01e057  "•"  ((DES_OUT(55 downto 48) xor DES_OUT(23 downto 16)) &
   Seaac9  "•"  ((DES_OUT(47 downto 40) xor DES_OUT(15 downto 8)) &
   44d11  "•"  ((DES_OUT(39 downto 32) xor DES_OUT(7 downto 0)) &
   7c4b5b  "•"  (DES_OUT(31 downto 0));

5ca5fa  dfa5a  cc6f48 when others =>.
   0abcd  "•" EXTRA_XOR_OUT <= DES_OUT;

7ca5fa  b924b  end case;

bc11f8  end process EXTRA_XOR_PR;

f69a82  EXTRA_SELECT <= SELECT1_DLY & EXTRA_XOR & USE_CBC;

f3590f  --EXTRA_SELECT <= SELECT1 & EXTRA_XOR & USE_CBC;

555356  610f89  begin
   ec6118  if RST_N = '0' then
      b83ec  STARTDES <= '0';
   8084bd  elsif CLK'event and CLK = '1' then
      a25435  "•"  STARTDES <= DONE or LOAD; ---·17 clocks
   fabe3f  "•"  STARTDES <= PRE_DONE or LOAD; ---·16 clocks
   6ba5fa  "•"

f2df08  end if;

7db3b0  end process STARTDES_PR;

f3af5a  14d499  KEY_INCR DECR PR: process (RST_N, CLK)

a15356  750f89  begin
   f26118  if RST_N = '0' then
      e50de  "•" KEY_INCR <= '0';
   28b39e  "•" KEY_DECR <= '0';
   951a84bd  elsif CLK'event and CLK = '1' then
      c7e055  "•"  KEY_INCR <= (CNT_GT_10 and not(DONE) and SEARCHING_DLY) and (
         935a62  "•"  not(MATCH) 'or' --- normal case
   61f7db  "•"  SELECT1 or 'or' --- false match
    7e5f79  "•"  FIRST_DES);
   db4587  "•"  KEY_DECR <= (CNT_GT_10 and not(DONE) and SEARCHING_DLY) and --- timing
   73bc59  "•"  (MATCH and not(SELECT1)) --- only backup if match on C0
   550e04  "•"  and not(FIRST_DES);
   49af5a  "•"

86df08  end if;
   0aee68  end process KEY_INCR DECR PR;

a9526  58f268  FALSE.MATCH <= '1' when (MATCH_DLY_CYCLE2 = '1') and (MATCH = '0') and (SEARCH
   9db53  "•"  ING_DLY = '1')
   e14767  "•"  else '0';

942b60  "•" timing block, sensitive to START
   fc49f8  "•" timing block, sensitive to START
   bc216e  "•"  else '0';

225f39  PRE_DONE <= '1' when (CNT = "00111") else '0';
   07bddd  RESET SEARCHING <= '1' when (CNT = "01100") else '0';
   f8ba5f  "•"

7b9367  "•"  CNT_EQ_1 <= '1' when (CNT = 1) else '0';
   684c89  "•"  CNT_EQ_10 <= '1' when (CNT > 10 and CNT < 10) else '0';

563599  "•"  CNT_EQ_10 <= '1' when (CNT = 10) else '0';

99228a  "•"  CNT_GT_10 <= '1' when (CNT > 10) else '0';

8a535a  "•"

e5c874  "•" SEARCHING PR: process (RST_N, CLK)
Chapter 6: Chip Source Code

--0906 0016131d0b28003000e Page 7 of search.vhd

b65356

b6f89 begin
286118 if RST_N = '0' then
5a14d if (LOAD = '1') or (SEARCHING = '1') then
4e3178 SEARCHING <= '0';
289c7f CLEAR_SEARCH <= '0';
3b8bd elsif CLK'event and CLK = '1' then
d5b815 SEARCHING <= SEARCHING;
78a5f a
61a1b search active
42d18 if (LOAD = '1') or (SEARCHING = '1') then
1d15d if SEARCHING <= '1';
82578d end if;
19a5f a
0b5fd found C1
5b1061 if (CLEAR_SEARCH_BAK = '1') then
e00969 SEARCHING <= '0';
b9570d end if;
7af5a
8073b
f52661 CLEAR_SEARCH <= CLEAR_SEARCH_BAK;
c0d0b end if;
69d83c
b2659d end process SEARCHING_PR;
5a85c if CHECKSAME_KEY = '1' when (KEY(1 downto 0) = KEY_ODD_DLY2) else '0';
74a5f a
b85356
5c5486 SELECT1_PR: process(RST_N,CLK)
645356
b6f89 begin
b76118 if RST_N = '0' then
6385d if SELECT1 = '1';
4384b elsif CLK'event and CLK = '1' then
79a5f a
1358c1 found C0, look for C1
3b54be if (MATCH = '1') and (SELECT1 = '0') and (PRE_DONE = '1') then
50e9d if SELECT1 <= '1';
16570d end if;
78a5f a
356ec8 Restart by PC or C1 is not a match
e56d5f if (SELECT1 = '1') and (PRE_DONE = '1') and (SEARCHING_DLY = '1') then
5b1061 ((MATCH = '1') and (PRE_DONE = '1') and (SEARCHING_DLY = '1')) then
9df1b6 SELECT1 <= '0';
3a570d end if;
75f3b4
7e2e23 if (PRE_DONE = '1') then
410eef SELECT1_DLY <= SELECT1;
b5f80d end if;
7dd85c
355784 end process SELECT1_PR;
1f1536
559ec2 SEARCH_OUT <= SEARCHING;
9acff LOAD <= SEARCH_DLY1 and PRE_DONE and not(SEARCH_DLY2); 17 clocks
18880e FIRST_Des <= SEARCH_DLY2 and not(SEARCH_DLY3);
79a5f 1 '1' when (MATCH = '1') and (SELECT1 = '0');
79a5f 1 and (SEARCHING = '1') and (RESET_SEARCHING = '1');
5ebed4 and (SEARCHING = '1')) else '0';
9a5f a
358fe SELECT ONE <= SELECT1;
daf11 KEY_OUT <= KEY;
54f7f8 DES_OUTPUT <= DES_OUT;
0f628a MATCH_OUT <= MATCH;
925356
3f0b8a end beh;
5a5356
61a5f a
---ed38 000a88fc985800300f Page 1 of start_re.vhd

bb97d --------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |

aa53a -- Author ---------------: Tom Vu --------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |

f066d -- Date ----------------: 09/19/97 --------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |--------------------------- |

7047d -- Description: UProcessor interface

407af library ieee;
611e9 use IEEE.std_logic_1164.all;
ba6f6 entity START_REG is

c4af5a port( 
RST_N : in std_logic;
c9858003000f

bc4c76 CHIP_EN : in std_logic;

6c5c79 START_REG : in std_logic;

83c7f6 RST_N_0 : in std_logic;

2c3455 WRB : in std_logic;

ae06c5 ADDRSEL2 : in std_logic;

27a88e ADDR : in std_logic_vector(7 downto 0);

d381d2 CLEAR_SEARCH : in std_logic_vector(23 downto 0);

b2a5f5 SEARCH_IN : out std_logic_vector(23 downto 0);

74198e DATAI : in std_logic_vector(7 downto 0)

c4737c );

a7af5a

a9af5a 77b995 end START_REG;

49af5a

c481be architecture beh of START_REG is

ca5356

2ca5f5

d6af5a

4be6ee signal SEARCH_IN_REG : std_logic_vector(23 downto 0);

bc4c76 signal SEARCH_RST_N_0 : std_logic;

6c5c79 signal SEARCH_RST_N_1 : std_logic;

38cc68 signal SEARCH_RST_N_2 : std_logic;

7a7c67 signal SEARCH_RST_N_3 : std_logic;

ca40c4a signal SEARCH_RST_N_4 : std_logic;

7d14c5 signal SEARCH_RST_N_5 : std_logic;

a4c54 signal SEARCH_RST_N_6 : std_logic;

6a3c5b signal SEARCH_RST_N_7 : std_logic;

88ec0e signal SEARCH_RST_N_8 : std_logic;

0f1dca signal SEARCH_RST_N_9 : std_logic;

6f95f5 signal SEARCH_RST_N_10 : std_logic;

2e114a signal SEARCH_RST_N_11 : std_logic;

ad86bb signal SEARCH_RST_N_12 : std_logic;

499b98 signal SEARCH_RST_N_13 : std_logic;

69738 signal SEARCH_RST_N_14 : std_logic;

913928 signal SEARCH_RST_N_15 : std_logic;

66a389 signal SEARCH_RST_N_16 : std_logic;

3ac519 signal SEARCH_RST_N_17 : std_logic;

cfc9e signal SEARCH_RST_N_18 : std_logic;

f0418e signal SEARCH_RST_N_19 : std_logic;

2a6af8 signal SEARCH_RST_N_20 : std_logic;

4af8e signal SEARCH_RST_N_21 : std_logic;

cbb7a signal SEARCH_RST_N_22 : std_logic;

d1356a signal SEARCH_RST_N_23 : std_logic;

deaf5af begin

b0f89 begin

a1af5a

df5356 SEARCH_IN0_PR : process(SEARCH_RST_N_0, WRB)

5b5356 begin

e0e9bc begin

a1a8c8 if (SEARCH_RST_N_0 = '0') then

ca0fed4 if (SEARCH_RST_N_0 = '0') then

1d5a3a if (SEARCH_RST_N_0 = '0') then

d9e9ed elsif (WRB'event and WRB = '1') then

07e9b2 if ((CHIP_EN = '1') and (ADDRSEL2 = '0') and (ADDR = "01000111")) then

a6f6ed if ((CHIP_EN = '1') and (ADDRSEL2 = '0') and (ADDR = "01000111")) then

5f523c else

cedece9 else

d9e9ed SEARCH_IN_REG(0) <= SEARCH_IN_REG(0);

5f523c else

cedece9 else

a862af end if; •
--7851 000ad1683558003000f Page 2 of start_re.vhd

5562af end if;
7da8c8 end process SEARCH_IN0_PR;
295356 -------------------------------
af1cdd SEARCH_IN1_PR: process(SEARCH_RST_N_1, WRB)
a45356 -------------------------------
a89bec begin ...
268b3e if (SEARCH_RST_N_1 = '0') then
890af : SEARCH_IN_REG(1) <= '0';
4aae0 else if (WRB'event and WRB = '1') then
6eec2 if ((CHIP_EN = '1') and (ADDR = "01010111")) then
34f87d : SEARCH_IN_REG(1) <= DATAI(0);
5462af end if;
7da8c8 end process SEARCH_IN1_PR;
fa584 SEARCH_IN2_PR: process(SEARCH_RST_N_2, WRB)
795356 -------------------------------
b59bec begin ...
40300 if (SEARCH_RST_N_2 = '0') then
ccf11c : SEARCH_IN_REG(2) <= '0';
17c8d elsif (WRB'event and WRB = '1') then
119768 if ((CHIP_EN = '1') and (ADDR = "01010111")) then
76625c : SEARCH_IN_REG(2) <= DATAI(0);
2d62af end if;
8a62af end if;
299bec begin ...
3e7a0 end process SEARCH_IN2_PR;
f95356 -------------------------------
fa584 SEARCH_IN3_PR: process(SEARCH_RST_N_3, WRB)
6b5356 -------------------------------
2d9bec begin ...
47ede if (SEARCH_RST_N_3 = '0') then
aa44d : SEARCH_IN_REG(3) <= '0';
21a0d elsif (WRB'event and WRB = '1') then
5892c5 if ((CHIP_EN = '1') and (ADDR = "01010111")) then
7c8c4 : SEARCH_IN_REG(3) <= DATAI(0);
b462af end if;
4362af end if;
2aa8c8 begin ...
1e2ba end process SEARCH_IN3_PR;
bc1356 -------------------------------
7eb261 SEARCH_IN4_PR: process(SEARCH_RST_N_4, WRB)
b25356 -------------------------------
af9bec begin ...
500dd if (SEARCH_RST_N_4 = '0') then
d046b : SEARCH_IN_REG(4) <= '0';
2dade elsif (WRB'event and WRB = '1') then
tfa18 if ((CHIP_EN = '1') and (ADDR = "01100111")) then
cb5af : SEARCH_IN_REG(4) <= DATAI(0);
e162af end if;
5462af end if;
99a8c8 begin ...
3a1356 end process SEARCH_IN4_PR;
593556 -------------------------------
123ee7 SEARCH_IN5_PR: process(SEARCH_RST_N_5, WRB)
0d5356 -------------------------------
e19bec begin ...
0738f if (SEARCH_RST_N_5 = '0') then
6851f : SEARCH_IN_REG(5) <= '0';
96ae0 else if (WRB'event and WRB = '1') then
7d405 if ((CHIP_EN = '1') and (ADDR = "0110111")) then
a2d8f : SEARCH_IN_REG(5) <= DATAI(0);
1362af end if;
ab62af end if;
ec1b8 begin ...
3ed97a end process SEARCH_IN5_PR;
45356 -------------------------------
3a37c SEARCH_IN6_PR: process(SEARCH_RST_N_6, WRB)
484536 -------------------------------
ad9bec begin ...
--a70b 000b004fd0803000f Page 3 of start_re.vhd

84f0b9 if (SEARCH_RST_N_6 = '0') then
5ea4f9 \ SEARCH_IN_REG(6) <= '0';
4ae0d elsif (WRB'event and WRB = '1') then
a0f42 if ((CHIP_EN = '1') and (ADDSLE2 = '0') and (ADDR = "01110111") then
84a3e \ SEARCH_IN_REG(6) <= DATAI(0);
952af end if;

b362af end if;

15a8c8

ef587 end process SEARCH_IN6_PR;

d5356

082ff SEARCH_IN7_PR: process(SEARCH_RST_N_7, WRB)
65356

2a9ebc begin ..
fa8e53 if (SEARCH_RST_N_7 = '0') then
d8f0d8 \ SEARCH_IN_REG(7) <= '0';
e3ae0d elsif (WB'event and WRB = '1') then
b5daef if ((CHIP_EN = '1') and (ADDSLE2 = '0') and (ADDR = "01111111") then
7bc42d \ SEARCH_IN_REG(7) <= DATAI(0);
4562af end if;

9a62af end if;

cb8c8

eb312c end process SEARCH_IN7_PR;

845356

6d42f SEARCH_IN8_PR: process(SEARCH_RST_N_8, WRB)
75356

829ebc begin ..
f811b7 if (SEARCH_RST_N_8 = '0') then
9ee694 \ SEARCH_IN_REG(8) <= '0';
0dae8d elsif (WB'event and WRB = '1') then
ef77fd if ((CHIP_EN = '1') and (ADDSLE2 = '0') and (ADDR = "10000111") then
bc26af \ SEARCH_IN_REG(8) <= DATAI(0);
4962af end if;

d52af end if;

3ca8c8

92bca5 end process SEARCH_IN8_PR;

765356

085a8 SEARCH_IN9_PR: process(SEARCH_RST_N_9, WRB)
275356

8a9ebc begin ..
0fd5d if (SEARCH_RST_N_9 = '0') then
d0038af \ SEARCH_IN_REG(9) <= '0';
58ae0d elsif (WB'event and WRB = '1') then
057230 if ((CHIP_EN = '1') and (ADDSLE2 = '0') and (ADDR = "10001111") then
44d8b9 \ SEARCH_IN_REG(9) <= DATAI(0);
0e82af end if;

ae62af end if;

36a8c8

7a08e end process SEARCH_IN9_PR;

0f5356

dabff6 SEARCH_IN10_PR: process(SEARCH_RST_N_10, WRB)
9b5356

e89ebc begin ..
c5b84 if (SEARCH_RST_N_10 = '0') then
e109db \ SEARCH_IN_REG(10) <= '0';
7b3ed8 elsif (WB'event and WRB = '1') then
35b97f if ((CHIP_EN = '1') and (ADDSLE2 = '0') and (ADDR = "10010111") then
fe8b9 \ SEARCH_IN_REG(10) <= DATAI(0);
5962af end if;

da62af end if;

f6a8c8

b6c353 end process SEARCH_IN10_PR;

fe5356

c592af SEARCH_IN11_PR: process(SEARCH_RST_N_11, WRB)
535356

3a8ebc begin ..
28c6e if (SEARCH_RST_N_11 = '0') then
e156af \ SEARCH_IN_REG(11) <= '0';
5b3e0d elsif (WB'event and WRB = '1') then
35b97f if ((CHIP_EN = '1') and (ADDSLE2 = '0') and (ADDR = "10011111") then
410514 \ SEARCH_IN_REG(11) <= DATAI(0);
0c62af end if;

bb62af end if;

-- VHD: 0001be5ca8003000f Page 4 of start_re.vhd

80a8c8 end process SEARCH_IN_11_PR;
35536 end process SEARCH_IN_12_PR:
1f5356 begin ..
754538 if (SEARCH_RST_N_12 = '0') then
46a2f1 SEARCH_IN_REG(12) <= '0';
1fae0d elsif (WRB'event and WRB = '1') then
373f5b7 if ((CHIP_EN = '1') and (ADDSEL2 = '0') and (ADDR = "10100111")) then
75f935 SEARCH_IN_REG(12) <= DATAI(0);
426a2f end if;
8e62af end if;
92a8c8 begin ..
80cb0 end process SEARCH_IN_12_PR;
1d5356 begin ..
85c9e6 SEARCH_IN_13_PR: process(SEARCH_RST_N_13, WRB)
915356 begin ..
a89e8c begin ..
d73bd if (SEARCH_RST_N_13 = '0') then
cc9f768 SEARCH_IN_REG(13) <= '0';
aeeaf0 elsif (WRB'event and WRB = '1') then
943a11 if ((CHIP_EN = '1') and (ADDSEL2 = '0') and (ADDR = "10101111")) then
321125 SEARCH_IN_REG(13) <= DATAI(0);
ba62af end if;
2a62af end if;
8c8a8c begin ..
c3e2f end process SEARCH_IN_13_PR;
145356 begin ..
c00836 SEARCH_IN_14_PR: process(SEARCH_RST_N_14, WRB)
9a5356 begin ..
cb9e8c begin ..
a4f5b5d if (SEARCH_RST_N_14 = '0') then
31578e SEARCH_IN_REG(14) <= '0';
fbaf0d elsif (WRB'event and WRB = '1') then
17415d if ((CHIP_EN = '1') and (ADDSEL2 = '0') and (ADDR = "10110111")) then
c2a356 SEARCH_IN_REG(14) <= DATAI(0);
4862af end if;
462af end if;
30a8c8 begin ..
78d5ff end process SEARCH_IN_14_PR;
855356 begin ..
a25c6 SEARCH_IN_15_PR: process(SEARCH_RST_N_15, WRB)
815356 begin ..
e99e8c begin ..
a7357 if (SEARCH_RST_N_15 = '0') then
5e021f SEARCH_IN_REG(15) <= '0';
e0ae0d elsif (WRB'event and WRB = '1') then
943a11 if ((CHIP_EN = '1') and (ADDSEL2 = '0') and (ADDR = "10111111")) then
492d76 SEARCH_IN_REG(15) <= DATAI(0);
db62af end if;
0d62af end if;
79a8c8 begin ..
9ad7d4 end process SEARCH_IN_15_PR;
195356 begin ..
5753a6 SEARCH_IN_16_PR: process(SEARCH_RST_N_16, WRB)
325356 begin ..
f19e8c begin ..
5db689 if (SEARCH_RST_N_16 = '0') then
10fcac SEARCH_IN_REG(16) <= '0';
69ae0d elsif (WRB'event and WRB = '1') then
9bf98d if ((CHIP_EN = '1') and (ADDSEL2 = '0') and (ADDR = "11000111")) then
5fb757 SEARCH_IN_REG(16) <= DATAI(0);
dd62af end if;
2a62af end if;
2a8c8 begin ..
8bd9a end process SEARCH_IN_16_PR;
375356 begin ..
c7726 SEARCH_IN_17_PR: process(SEARCH_RST_N_17, WRB)
325356 begin ..
e39e8c begin ..
e1c9e6 if (SEARCH_RST_N_17 = '0') then

Chapter 6: Chip Source Code
```vhdl
--f11c 001f74fa4a58003000f Page 6 of start_re.vhd

c1636b end process SEARCH_IN22_PR;
ec5356
81d911 SEARCH_IN23_PR: process (SEARCH_RST_N_23, WRB)
245356
519ebc begin ..
54e3a4 if (SEARCH_RST_N_23 = '0') then
127086 elsif (WRB'event and WRB = '1') then
61cae0 if ((CHIP_EN = '1') and (ADDSEL2 = '0') and (ADDR = "11111111")) then
bb2134 SEARCH_IN_REG(23) <= DATA(0);
4962af end if;
8262af end if;

8ca8c8
586740 end process SEARCH_IN23_PR;
8ca8c8

b435b2 SEARCH_RST_N_0 <= RST_N and not(CLEAR_SEARCH(0));
0a943e SEARCH_RST_N_1 <= RST_N and not(CLEAR_SEARCH(1));
0b7ebb SEARCH_RST_N_2 <= RST_N and not(CLEAR_SEARCH(2));
4adf37 SEARCH_RST_N_3 <= RST_N and not(CLEAR_SEARCH(3));
0ba3a0 SEARCH_RST_N_4 <= RST_N and not(CLEAR_SEARCH(4));
fc022c SEARCH_RST_N_5 <= RST_N and not(CLEAR_SEARCH(5));
34e8a9 SEARCH_RST_N_6 <= RST_N and not(CLEAR_SEARCH(6));
d34925 SEARCH_RST_N_7 <= RST_N and not(CLEAR_SEARCH(7));
811187 SEARCH_RST_N_8 <= RST_N and not(CLEAR_SEARCH(8));
a1b000 SEARCH_RST_N_9 <= RST_N and not(CLEAR_SEARCH(9));
b5e1a0 SEARCH_RST_N_10 <= RST_N and not(CLEAR_SEARCH(10));
1f9aa9 SEARCH_RST_N_12 <= RST_N and not(CLEAR_SEARCH(12));
940b25 SEARCH_RST_N_13 <= RST_N and not(CLEAR_SEARCH(13));
0e77b2 SEARCH_RST_N_14 <= RST_N and not(CLEAR_SEARCH(14));
86d63e SEARCH_RST_N_15 <= RST_N and not(CLEAR_SEARCH(15));
e33ebd SEARCH_RST_N_16 <= RST_N and not(CLEAR_SEARCH(16));
4a9d37 SEARCH_RST_N_17 <= RST_N and not(CLEAR_SEARCH(17));
0ec595 SEARCH_RST_N_18 <= RST_N and not(CLEAR_SEARCH(18));
fb6419 SEARCH_RST_N_19 <= RST_N and not(CLEAR_SEARCH(19));
f32ef SEARCH_RST_N_20 <= RST_N and not(CLEAR_SEARCH(20));
ea9363 SEARCH_RST_N_21 <= RST_N and not(CLEAR_SEARCH(21));
8c79e6 SEARCH_RST_N_22 <= RST_N and not(CLEAR_SEARCH(22));
bed8a SEARCH_RST_N_23 <= RST_N and not(CLEAR_SEARCH(23));
8ad83c SEARCH_RST_N <= SEARCH_IN_REG;
bbf5a
19b08a end beh;
2d5356
f7af5a
```

Chapter 6: Chip Source Code
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity TOP is
    port(
        CLK: in std_logic;
        RST_N: in std_logic;
        BOARD_EN: in std_logic;
        ALE: in std_logic;
        ADDSEL: in std_logic;
        WRB: in std_logic;
        RDB: in std_logic;
        ADDSEL2: in std_logic;
        AA_IN: in std_logic;
        ADDR: in std_logic_vector(7 downto 0);
        CHIP_ID: in std_logic_vector(7 downto 0);
        AA_OUT: out std_logic;
        CHIP_AA_OUT: out std_logic;
        DATA: inout std_logic_vector(7 downto 0);
        C0: in std_logic_vector(63 downto 0);
    );
end TOP;

architecture beh of TOP is
begin
    process(CLK, RST_N)
    begin
        if RST_N = '0' then
            SHIFT_REG <= DATA8_ARRAY;
        else
            if rising_edge(CLK) then
                if RST_N = '0' then
                    SHIFT_REG <= DATA8_ARRAY;
                else
                    case SELECT_ONE is
                        when '0' =>
                            SHIFT_REG <= DATA8_ARRAY;
                        when '1' =>
                            SHIFT_REG <= ADDR; -- std_logic_vector(23 downto 0);
                    end case;
                end if;
            end if;
        end if;
    end if;
end process;

end beh;

library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity TOP is
    port(
        CLK: in std_logic;
        RST_N: in std_logic;
        BOARD_EN: in std_logic;
        ALE: in std_logic;
        ADDSEL: in std_logic;
        WRB: in std_logic;
        RDB: in std_logic;
        ADDSEL2: in std_logic;
        AA_IN: in std_logic;
        ADDR: in std_logic_vector(7 downto 0);
        CHIP_ID: in std_logic_vector(7 downto 0);
        AA_OUT: out std_logic;
        CHIP_AA_OUT: out std_logic;
        DATA: inout std_logic_vector(7 downto 0);
        C0: in std_logic_vector(63 downto 0);
    );
end TOP;

architecture beh of TOP is
begin
    process(CLK, RST_N)
    begin
        if RST_N = '0' then
            SHIFT_REG <= DATA8_ARRAY;
        else
            if rising_edge(CLK) then
                if RST_N = '0' then
                    SHIFT_REG <= DATA8_ARRAY;
                else
                    case SELECT_ONE is
                        when '0' =>
                            SHIFT_REG <= DATA8_ARRAY;
                        when '1' =>
                            SHIFT_REG <= ADDR; -- std_logic_vector(23 downto 0);
                    end case;
                end if;
            end if;
        end if;
    end if;
end process;

end beh;
Chapter 6: Chip Source Code

-- acfd 0006a1758a80030010 Page 2 of top.vhd

e376cb ... C1 ... in std_logic_vector(63 downto 0);
e33ca4 ... SEARCH_OUT ... out std_logic;
706b53 ... CLEAR_SEARCH ... out std_logic;
cfc5f4 ... SELECT_ONE ... out std_logic;
1e0f8f ... DATAO ... out std_logic_vector(7 downto 0);
47737c ... );
6daf5a cfe2c6 end component;
8daf5a

6a2f5a

b76102 component UPI
3c0a36 port ( RST_N ... in std_logic;
e91a55 ... BOAR DH EN ... in std_logic;
3a2b30 ... ALE ... in std_logic;
485e97 ... ADDSEL1 ... in std_logic;
a26737 ... MRB ... in std_logic;
d08768 ... RDB ... in std_logic;
c13782 ... ADDSEL2 ... in std_logic;
e64c8d ... AA_IN ... in std_logic;
c3b4e7 ... ADDR ... in std_logic_vector(7 downto 0);
976ad7 ... CHIP_ID ... in std_logic_vector(7 downto 0);
3d053a ... SELECT_ONE ... in std_logic_vector(23 downto 0);
118f32 ... SEARCH_IN ... out std_logic_vector(23 downto 0);
42be0c ... SEARCH_OUT ... out std_logic_vector(23 downto 0);
7a8f53 ... CLEAR_SEARCH ... in std_logic;
9e6aef ... EXTRA_XOR ... out std_logic;
96a049 ... USE_CBC ... out std_logic;
b5f367 ... CHIP_AA_OUT ... out std_logic;
4bea5e ... AA_OUT ... out std_logic;
2aaf5a

03af5a

68596c ... PT_XOR_MASK ... out std_logic_vector(63 downto 0);
8a457 ... PT_BYTE_MASK ... out std_logic_vector(63 downto 0);
4e186 ... PT_VECTOR ... out std_logic_vector(255 downto 0);
d4f79c ... C0 ... out std_logic_vector(63 downto 0);
dc3889 ... C1 ... out std_logic_vector(63 downto 0);
63ccbc ... ADDR_KEY0 ... out std_logic_vector(6 downto 0);
6da1b1 ... ADDR_KEY1 ... out std_logic_vector(6 downto 0);
f7763f ... ADDR_KEY2 ... out std_logic_vector(6 downto 0);
d03645 ... ADDR_KEY3 ... out std_logic_vector(6 downto 0);
9bd132 ... ADDR_KEY4 ... out std_logic_vector(6 downto 0);
2b9c48 ... ADDR_KEY5 ... out std_logic_vector(6 downto 0);
b94bc6 ... ADDR_KEY6 ... out std_logic_vector(6 downto 0);
8f86bc ... ADDR_KEY7 ... out std_logic_vector(6 downto 0);
e59739 ... ADDR_KEY8 ... out std_logic_vector(6 downto 0);
0d443 ... ADDR_KEY9 ... out std_logic_vector(6 downto 0);
1057bd ... ADDR_KEY10 ... out std_logic_vector(6 downto 0);
765102 ... ADDR_KEY11 ... out std_logic_vector(6 downto 0);
55523 ... ADDR_KEY12 ... out std_logic_vector(6 downto 0);
3d57c ... ADDR_KEY13 ... out std_logic_vector(6 downto 0);
874d41 ... ADDR_KEY14 ... out std_logic_vector(6 downto 0);
7b4bf ... ADDR_KEY15 ... out std_logic_vector(6 downto 0);
b093 ... ADDR_KEY16 ... out std_logic_vector(6 downto 0);
4d4680 ... ADDR_KEY17 ... out std_logic_vector(6 downto 0);
986245 ... ADDR_KEY18 ... out std_logic_vector(6 downto 0);
4664fa ... ADDR_KEY19 ... out std_logic_vector(6 downto 0);
438033 ... ADDR_KEY20 ... out std_logic_vector(6 downto 0);
89868c ... ADDR_KEY21 ... out std_logic_vector(6 downto 0);
1f8d4d ... ADDR_KEY22 ... out std_logic_vector(6 downto 0);
f085f2 ... ADDR_KEY23 ... out std_logic_vector(6 downto 0);
81a9f7 ... DATA1 ... out std_logic_vector(7 downto 0);
88a365 ... DATAO ... out std_logic_vector(7 downto 0);
61737c ... );
8aaa5a cfe2c6 end component;
79af5a

aa0f89 begin
b29114 UPI0: UPI
f344b port map ( RST_N ... RST_N, BOARD_EN ... BOARD_EN, ALE ... ALE,
ADDSEL2 \rightarrow ADDR-KEY23 \rightarrow ADDR-KEY1 \rightarrow ADDR-KEY8 \rightarrow SEARCH-OUT \rightarrow DATAI \rightarrow WRB \rightarrow PT-BYTE-MASK \rightarrow SEARCH-IN \rightarrow RDB \rightarrow ADDR-KEY14 \rightarrow ADDR-KEY20 \rightarrow ADDR-KEY13 \rightarrow SELECT-ONE \rightarrow AA-IN \rightarrow SEARCH \rightarrow DATAO \rightarrow USE-CBC \rightarrow ADDR-KEY7 \rightarrow ADDR-KEY21 \rightarrow C1 \rightarrow ADDR-KEY \rightarrow RST-N \rightarrow ADDR \rightarrow CLEAR-SEARCH \rightarrow ADDR-KEY9 \rightarrow RDB \rightarrow ADDR-KEY22 \rightarrow ADDR-KEY2 \rightarrow \cdots \rightarrow SEARCH-OUT \rightarrow DATAI \rightarrow ADDR-KEY12 \rightarrow EXTRA-XOR \rightarrow ADDR-KEY3 \rightarrow C0 \rightarrow ADDR-KEY6 \rightarrow ADDR-KEY18 \rightarrow ADDR-KEY10 \rightarrow WRB \rightarrow ADDR-KEY5 \rightarrow C1 \rightarrow ADDR-KEY0 \rightarrow ADDR-KEY11 \rightarrow ADDR-KEY17 \rightarrow ADDR-KEY19 \rightarrow ADDR-KEY1 \rightarrow ADDR-KEY7 \rightarrow ADDR-KEY15 \rightarrow ADDR-KEY16 \rightarrow ADDR-KEY14 \rightarrow ADDR-KEY13 \rightarrow ADDR-KEY12 \rightarrow ADDR-KEY11 \rightarrow ADDR-KEY10 \rightarrow ADDR-KEY9 \rightarrow ADDR-KEY8 \rightarrow ADDR-KEY7 \rightarrow ADDR-KEY6 \rightarrow ADDR-KEY5 \rightarrow ADDR-KEY4 \rightarrow ADDR-KEY3 \rightarrow ADDR-KEY2 \rightarrow ADDR-KEY1 \rightarrow ADDR-KEY0 \rightarrow ADDR-KEY1 \rightarrow ADDR-KEY2 \rightarrow ADDR-KEY3 \rightarrow ADDR-KEY4 \rightarrow ADDR-KEY5 \rightarrow ADDR-KEY6 \rightarrow ADDR-KEY7 \rightarrow ADDR-KEY8 \rightarrow ADDR-KEY9 \rightarrow ADDR-KEY10 \rightarrow ADDR-KEY11 \rightarrow ADDR-KEY12 \rightarrow ADDR-KEY13 \rightarrow ADDR-KEY14 \rightarrow ADDR-KEY15 \rightarrow ADDR-KEY16 \rightarrow ADDR-KEY17 \rightarrow ADDR-KEY18 \rightarrow ADDR-KEY19 \rightarrow ADDR-KEY20 \rightarrow ADDR-KEY21 \rightarrow ADDR-KEY22 \rightarrow ADDR-KEY23 \rightarrow ADDR-KEY24 \rightarrow DATAI \rightarrow DATAO \rightarrow DATAO
01f37c \cdots ;
a3a15a
2b5356
fa1c46 gen0: for i in 0 to 23 generate 361b1f SEARCH_UNITX: SEARCH_UNIT
c72b2b port map(CLK \rightarrow CLK, 57a41e \rightarrow RST_N, 52a27 \rightarrow WRB, b87e43 \rightarrow RDB, 0aba0c \rightarrow PTBYTE_Mask, ecda55 \rightarrow SEARCH, 17ba2a \rightarrow SELECT_ONE, c964e3 \rightarrow ADDR_KEY, 2f1f94 \rightarrow EXTRA_XOR, 7b93d0 \rightarrow USE_CBC
4b0cc1 \rightarrow DATAI, 6bc7fa \rightarrow PT_VECTOR, 40f8f1 \rightarrow C0, eb5c94 \rightarrow C1, fd8a2a \rightarrow SEARCH_OUT, 2e5edf \rightarrow DATAO, 5d737c \cdots ;
--6a27 0012f356e1f80030010 Page 4 of top.vhd

```vhdl
fc7522  end generate;
755356  
722595  • DATAI <= DATA;
6b598a  • DATAI <= DATAO when (RDB = '0' and ADDSEL2 = '0') else others => 'Z';
92b08a  end beh;
f05356  
17af5a
5aaf5a
2faf5a
```
library ieee;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity UProcessor is
    port( RST_N : in std_logic;
          BOARD_EN : in std_logic;
          ALE : in std_logic;
          ADDSEL : in std_logic;
          RDB : in std_logic;
          f2000c5 : in std_logic;
          AA_IN : in std_logic;
          ADDR : in std_logic_vector(7 downto 0);
          ec8827 : in std_logic_vector(7 downto 0);
          70b08d : in std_logic_vector(23 downto 0);
          30c364 : in std_logic_vector(23 downto 0);
          5bf05b : in std_logic_vector(23 downto 0);
    );
    begin
end UProcessor;
Chapter 6: Chip Source Code

-- 9953 00085f7c1da80030011 Page 2 of upi.vhd

b11f4 signal SEARCH_IN_BAK : std_logic_vector(23 downto 0);
cf1b46 signal CHIP_EN : std_logic;
a9af5a component ADDR_KEY
2b14be port (
 943782 ADDSEL2 in std_logic;
d5748f CHIP_EN in std_logic;
3eb47e ADDR in std_logic_vector(7 downto 0);
c8af5a 83 eccb ADDR KEY0 out std_logic_vector(6 downto 0);
c9a1b1 ADDR KEY1 out std_logic_vector(6 downto 0);
59af5a ADDR KEY2 out std_logic_vector(6 downto 0);
903b45 ADDR KEY3 out std_logic_vector(6 downto 0);
7bd132 ADDR KEY4 out std_logic_vector(6 downto 0);
449c48 ADDR KEY5 out std_logic_vector(6 downto 0);
684bc6 ADDR KEY6 out std_logic_vector(6 downto 0);
9580bc ADDR KEY7 out std_logic_vector(6 downto 0);
2c9739 ADDR KEY8 out std_logic_vector(6 downto 0);
58ad43 ADDR KEY9 out std_logic_vector(6 downto 0);
b157bd ADDR KEY10 out std_logic_vector(6 downto 0);
8a5102 ADDR KEY11 out std_logic_vector(6 downto 0);
645ac3 ADDR KEY12 out std_logic_vector(6 downto 0);
785c7c ADDR KEY13 out std_logic_vector(6 downto 0);
84cb34 ADDR KEY14 out std_logic_vector(6 downto 0);
a3f1fe ADDR KEY15 out std_logic_vector(6 downto 0);
ab403f ADDR KEY16 out std_logic_vector(6 downto 0);
2b4f80 ADDR KEY17 out std_logic_vector(6 downto 0);
446245 ADDR KEY18 out std_logic_vector(6 downto 0);
518f4a ADDR KEY19 out std_logic_vector(6 downto 0);
378033 ADDR KEY20 out std_logic_vector(6 downto 0);
62868c ADDR KEY21 out std_logic_vector(6 downto 0);
128d4d ADDR KEY22 out std_logic_vector(6 downto 0);
88a27f ADDR KEY23 out std_logic_vector(6 downto 0);
85737c);
3ae2c6 end component;
a4d83c 94af5a
5232a1 component REG_RDWR
3dcba7 port ( RST_N in std_logic;
bef1a5 BOARD_EN in std_logic;
58db38 ALE in std_logic;
d56f97 ADDSEL1 in std_logic;
0d673f WRB in std_logic;
f19768 RDB in std_logic;
213782 ADDSEL2 in std_logic;
d1e4cd AA_IN in std_logic;
d5b47e ADDR in std_logic_vector(7 downto 0);
8b6ad7 CHIP_EN in std_logic_vector(7 downto 0);
9b0b7f SEARCH_OUT in std_logic_vector(23 downto 0);
f3053a SELECT_ONE in std_logic_vector(23 downto 0);
bdff73 SEARCH_IN in std_logic_vector(23 downto 0);
9f1f5a
25bcc9 CHIP_EN in std_logic;
b9ea5e AA_OUT in std_logic;
de4a20 CHIP_AA_OUT in std_logic;
606eaf EXTRA_XOR in std_logic;
43a849 USE_CBC in std_logic;
b3596c PT_XOR_MASK in std_logic_vector(63 downto 0);
a1a457 PT_BYTE_MASK in std_logic_vector(7 downto 0);
a8e186 PT VECTOR in std_logic_vector(255 downto 0);
377f79c C0 in std_logic_vector(63 downto 0);
2d3889 C1 in std_logic_vector(63 downto 0);
774d79 DATA1 in std_logic_vector(7 downto 0);
86d0f9 DATA0 in std_logic_vector(7 downto 0);
5c737c);
15e2c6 end component;
35af5a 23d83c
69d9b9 component - START_REG
becba7 port ( RST_N in std_logic;
85748f CHIP_EN in std_logic;
56c737       WRB          : in std_logic;
dd3782       ADDSEL2     : in std_logic;
5cb47e       ADDR         : in std_logic_vector(7 downto 0);
59b1d2       CLEAR_SEARCH : in std_logic_vector(23 downto 0);
b9af5a       SEARCH_IN    : OUT std_logic_vector(23 downto 0);
1b25bf       DATAI        : in std_logic_vector(7 downto 0);
ca737c       end component;

3b0f89 begin

375356       ADDR_KEYX : ADDR_KEY

8e5b8 ADDSEL2 ADD_KEY0          : => ADDR_KEY0,
341404 CHIP_EN              => CHIP_EN,
659c40 ADDR           : => ADDR,
b8af5a ADDR_KEY1 ADDR_KEY11  : => ADDR_KEY11,

13af0 REG_RDWRY : REG_RDWR

41af5a end component;
--25a4 0013211398f80030011 Page 4 of upi.vhd

789938 START_REGX : START_REG
6cb1c0 port map(RST_N ........ => RST_N,
7d1404 ...... CHIP_EN ........ => CHIP_EN,
0414b5 ...... WRB ........ => WRB,
f1c99d ...... ADDSEL2 ........ => ADDSEL2,
b9c40 ...... ADDR ........ => ADDR,
600130 ...... CLEARSEARCH .... => CLEARSEARCH,
38af5a ...... SEARCH_IN ...... => SEARCH_IN_BAK,
2a6bf ...... DATAI .......... => DATAI

d7737c ....);
ec689ea SEARCH_IN <= SEARCH_IN_BAK;
ecb08a end beh;
c95356 ----------------------------------
32af5a
This chapter contains C-language software that simulates the operation of the custom DES Cracker chip. This software is useful for showing people how the chip works, and to make test-vectors to let machines determine whether chips are properly fabricated.

We wrote this simulator before the chip was designed, to explore different design ideas. It should produce results identical to the final chips. We designed it for clarity of description, and flexibility in trying out new ideas, rather than speed. If you don't understand how the chip works, you can try some experiments by building this software on an ordinary PC or Unix machine with an ordinary C compiler, such as Borland C++ 3.1.

Building physical chips is an error-prone process. Each chip might be contaminated by dust or flaws in the silicon materials. There's no way to tell whether a given chip will work or not, without trying it out. So chip-building companies require that when you design a chip, you also provide test vectors. These list the voltages to put on each input pin on the chip, and how the chip-testing machine should vary them over time. The vectors also specify exactly what output signals the chip-tester should be able to measure on the chip's output pins. If the chip tester feeds all the input signals to the chip, step by step, and sees all the corresponding output signals, the chip "passes" the test. If any output signals differ from the specification, the chip "fails" the test and is discarded.

Passing such a test doesn’t prove that a chip has been fabricated correctly. It only proves that the chip can run the small set of tests that the designer provided. Creating test vectors which verify all parts of a chip is an art. The expense of testing a chip is proportional to the size of the tests, so they are usually short and direct. Thus, they also act as small examples that you can use to explore your understanding of how the chip works.

Chapter 4, Scanning the Source Code, explains how to read or scan in these documents.
Chapter 7: Chip Simulator Source Code

---9a44 0014b4364e180040001 Page 1 of MANIFEST

7bf681  1  MANIFEST
63b635  2  README
476ecc  3  blaze.scr
8a69aa  4  cbc1.scr
581046  5  cbc2.scr
1feade  6  cbc3.scr
868e50  7  des.c
f3db2a  8  des.h
bbf31a  9  ecb.scr
039f0d 10  mini.scr
02ce39 11  misc.c
60fc96 12  misc.h
b51b5d 13  random.scr
cfb60c 14  ref.c
aa84bd 15  sim.c
0beac5 16  sim.h
4b2104 17  testvec.c
Chapter 7: Chip Simulator Source Code

---

-926a 0011c402f8d80040002 Page 1 of README

e0af5a
3f1c37 testvec.c (compile with sim.c and des.c): - Generates and runs test
9a0001 vectors. - This program will both run existing input vectors, or,
08239f generate new ones (either randomly or from a script). - When compiled
c43481 under DOS, it can either produce Unix (LF only) or DOS (CR/LF)
37d2c3 output files (select with the RAW parameter).
a1af5a
26a7b8 To run the ecb.scr sample script and:
473eef - Store test vectors which go to the chip in TOCHIP.EXT
5b66ea - Store test vectors received from the chip in FROMCHIP.EXT,
0763b5 - Produce Unix-style output (LF only)
9e1741 - Store debugging output in debug.out.
10af5a
f46e57 rm *.EXT
a1f54e - Store test vectors which go to the chip in TOCHIP.EXT
f46e57 FROMCHIP.EXT RAW < ecb.scr > debug.out
c7ed9a 1ff74b If TOCHIP.EXE already exists when the program is run, it will
1ff74b read it (instead of expecting a script from stdin).
917018
60af5a
b4a916 Use the script random.scr to produce a random test vector, e.g.:
b7bdc0 - Store test vectors which go to the chip in TOCHIP.EXT
4eaf5a RAW < random.scr > debug.out
52af5a
d7ecf1 ref.c (compile with des.c misc.c): - Runs test scripts (.scr files)
0b7e68 and prints any keys that match. - This is basically a stripped-down
abd91f test vector generator for debugging purposes. - (It doesn't make any
9749f1 attempt to match timings.)
2caf5a
1caf5a
Chapter 7: Chip Simulator Source Code

--a854 001ab3b125780040003 Page 1 of blaze.scr

95a107 00 01 02 03 04 05 06 07 10 11 12 13 14 15 16 1720 21 22 23 24 25 26 2730 31 32
2e79b3 33 34 35 36 3740 41 42 43 44 45 46 4750 51 52 53 54 55 56 57
159ec4 0000000000000000 00 XOR MASK
7e10d0 123456789ABCDEF0 00 Cipher text 0
690908 123456789ABCDEF0 Cipher text 1
9cb374 0F Plain text byte mask
ebed5e 0 0 use CBC
72af2d 1 extra XOR
a25b0d 1 don't seed PRNG (use this input file)
fa2fcc 01020304050607 starting key
2b6f6b 8000 number of clocks
f8af5a 49af5a
29c4d8 d6 e9 89 fa 'DES_DECRYPT(k=00020304050607, c=123456789ABCDEF0)=B8 C0 1B 3E 35
84c98d DB 2F DE 00 B1 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14
b1422d 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
89c4e2 F3 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49
f7e1db 4A 4B 4C 4D 4E 4F50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64
58e46f 65 66 67 68 69 6A 6B 6C 6D 6E 6F70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
f10795 F8 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F90 91 92 93 94 95 96 97 98 99
505b42 9A 9B 9C 9D 9E 9F .A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AFB0 B1 B2 B3 B4
7fa3b9 4B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE
f6f44d .CF00 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9
a8d9b 9 EA EB EC ED EE EF0 F1 F2 F3 F4 F5 F6 F7 F8 FA FB FC FD FE
aba5f5a
Chapter 7: Chip Simulator Source Code

--- a728 0015c860f9980040004 Page 1 of cbc1.scr

f64ce1 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1E
b53734 A1 B1 C1 D1 E1 F0 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34
5f0653 .35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E
bc6d97 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69
a1cb67 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7
2b3164 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7
babb7d B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2
54b467 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
2f281f 37393b51def84190 "' XOR MASK
1810d0 123456789ABCDEF0 "' Cipher text 0
59ca3f 0102030405060708 "' Cipher text 1
b98c19 00 "' Plain text byte mask
56ccd1 1 "' use CBC
b084f 0 "' extra XOR
095b0d 1 "' don't seed PRNG (use this input file)
322fc0 01020304050607 "' starting key
359df9 10000 "' number of clocks
5caf5a
Chapter 7: Chip Simulator Source Code

--a112 00101ebc6db80040005 Page 1 of cbc2.scr

f64ce1 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 0b8b2a C ED EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 56ac5f 42341231234123f ... XOR MASK 0b327c 0000000000000000 ... Ciphertext 0 ee53f1 1578345691832465 ... Ciphertext 1 23e767 04 ............... Plaintext byte mask 20cdad 1 ............... use CBC 00d84f 0 ............... extra XOR 795b0d 1 ............... don't seed PRNG (use this input file) 12de95 FFFFFFFFFFFFFFF0 ....... starting key 309df9 10000 ............... number of clocks 01af5a
Chapter 7: Chip Simulator Source Code

---3fb8 001348ab9e68004000 Page 1 of cbc3.scr

2c9f3b 00 01 02 03 04 05 07 08 09 0D 0E 0F 10 11 12 14 15 17 1A 1B 1C 1D 1F 20 21 24 28
bd57b8 5 28 29 2A 2B 2C 2E 30 31 32 35 36 37 39 3A 3C 3D 3E 40 42 43 44 45 48 49 4A 4B
1b6fb2 4C 4F 50 51 53 54 56 57 58 59 5C 5D 5F 61 62 63 64 66 67 69 6B 6C 6D 6F 70 71
6e5f88 72 73 77 78 7A 7B 7D 7E 7F 80 82 86 87 89 8A 8B 8C 8D 90 92 93 94 95 97 98 99
753dfe 9 9A 9B 9E 9F A0 A2 A3 A4 A5 A6 A8 AA AC AD AE AF B0 B1 B3 B4 B7 B8 B9 F8 F9 FA
958c0a FB FC FD FF BB BC BD BE C0 C1 C3 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2 D3
09fa1d D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF E0 E1 E2 E3 E4 # E5 E6 E7 E8 E9 EA EB EC ED
473f52 EE EF F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF
638b27 0124801248012480---- XOR MASK
3745a7 FFFFFFFFFFFFFF... Cipher text 0
1d2a4 000000000000000... Cipher text 1
d6b12c 80 .......................... Plaintext byte mask
72cda 1 .......................... use CBC
37d84f 0 .......................... extra XOR
e85b0d 1 .......................... don't seed PRNG (use this input file)
afa481 000000000000000... starting key
e49d19 10000 .......................... number of clocks
34af5a
---b787 000d22ad6f780040007 Page 1 of des.c

8d2d03 /* ********************************************* */
d72f9b /* */
b27b4d /* Software Model of ASIC DES Implementation. */
49f39b /* */
9f9e08 /* Written by Paul Kocher, Tel: 415-397-0111, Email: paul@cryptography.com */
ce29eb /* */
e7489b /* ********************************************* */
de29eb /* */
c51ecb /* IMPLEMENTATION NOTES: */
3429eb /* */
987682 /* This DES implementation adheres to the FIPS PUB 46 spec and produces */
a90da /* standard output. The internal operation of the algorithm is quite */
9a876e /* different from the FIPS. For example, bit orderings are reversed */
a14be6 /* (the right-hand bit is now labelled as bit 0), the S tables have */
06b9c7 /* rearranged to simplify implementation, and several permutations have */
4666d /* been inverted. No performance optimizations were attempted. */
1289eb /* */
3e489b /* ********************************************* */
aa29eb /* */
496eef /* */
ba29eb /* */
0bc443 /* Version 1.0: Initial release -- PCK. */
d7b74c /* Version 1.1: Altered DecryptDES exchanges to match EncryptDES. -- PCK */
fa5c77 /* Version 1.2: Minor edits and beautifications. -- PCK */
e8d8c3 /* ********************************************* */
9aaf5a /* */
a8a5f5a /* */
005far2b /* include <stdio.h> */
a3ba3 /* include <stdlib.h> */
94324c /* include <string.h> */
e92bac /* include "des.h" */
fb5a5a /* */
7f7461 static void ComputeRoundKey(bool roundKey[56], bool key[56]);
6d84a3 static void RotateRoundKeyLeft(bool roundKey[56]);
57c5f3a static void RotateRoundKeyRight(bool roundKey[56]);
741504 static void ComputeIP(bool LR[32], bool RC[32], bool inBlk[E43],
1c87da static void ComputeIP(bool outBlk[E43], bool LR[32], bool RC[32]);
fe07fb static void ComputeF(bool outf[32], bool RC[32], bool roundKey[56]);
e9f94ee static void ComputeP(bool output[32], bool input[32]);
71fffe static void ComputeS_Lookup(int k, bool output[64], bool input[64]);
25a51a3 static void ComputePC2(bool subkey[E43], bool roundKey[56]);
5df9d9 static void ComputeExpansion(bool expandedBlock[48], bool RC[32]);
662f5b static void DumpBin(char *str, bool *b, int bits);
a45bc static void Exchange_Land_R(bool LR[32], bool RC[32]);
39a5f5a /* */
efc223 int EnableDumpBin = 0;
f5a5f5a /* */
7caf5a /* */
844a5a /* */
d84d6c /* ********************************************* */
a5c68f /* */
033c1a /* DES TABLES */
1c6ac /* */
264d6c /* ********************************************* */
6fa5f5a /* */
76a5f5a /* */
85385 /* */
83385 /* */
aa55a /* IP: Output bit table, DES_IP[i] equals input bit i. */
3045d /* */
09c166 static int table_DES_IP[E43] = ( 829d69 ... 39, 7, 47, 15, 55, 23, 63, 31, 
10e827 ... 38, 6, 46, 14, 54, 22, 62, 30, 
4b38ae ... 37, 5, 45, 13, 53, 21, 61, 29, 
2b6de0 ... 36, 4, 44, 12, 52, 20, 60, 28, 
5b247 ... 35, 3, 43, 11, 51, 19, 59, 27, 
c7e709 ... 34, 2, 42, 10, 50, 18, 58, 26, 
28829e ... 33, 1, 41, .9, 49, 17, 57, 25, 
4811f ... 32, .0, 40, .8, 48, 16, 56, 24, 
2882f7 );
39a5f5a /* */
95a5f5a /* */
d328e5 /* */
--43e9 000122426b0000007 Page 2 of des.c

ac48ca */ * FP: Output bit table DES_FP[i] equals input bit i.
c0495d */

2dde2a static int table DES_FP[64] = {
  675b71 ...... 57, 49, 41, 33, 25, 17, .9, .1,
  768cd9 ...... 59, 51, 43, 35, 27, 19, 11, .3,
  b2996 ...... 61, 53, 45, 37, 29, 21, 13, .5,
  935b71 ...... 63, 55, 47, 39, 31, 23, 15, .7,
  504e3f ...... 65, 48, 40, 32, 24, 16, .8, .0,
  62d997 ...... 70, 52, 44, 36, 28, 20, 12, .4,
  f5cd8 ...... 68, 50, 42, 34, 26, 18, 10, .2,
  64da9 ...... 62, 54, 46, 38, 30, 22, 14, .6
2d82f7);
7ba5a
b04f5a
0f3e5 / *
85d05 /* PC1: Permutation choice 1, used to pre-process the key
da495d */
a40c38 static int table DES_PC1[56] = {
  0db9e ...... 27, 19, 11, 31, 39, 47, 55,
  bb28e ...... 26, 18, 10, 30, 38, 46, 54,
  cf8d2c ...... 25, 17, 9, 29, 37, 45, 53,
  ce1d6 ...... 24, 16, 8, 28, 36, 44, 52,
  62bf91 ...... 23, 15, 7, -5, 35, 43, 51,
  d72feb ...... 22, 14, -6, 2, 34, 42, 50,
  b491e ...... 21, 13, -5, -1, 33, 41, 49,
  b0d02f ...... 20, 12, -4, -6, 32, 40, 48
1782f7);
67a5a
16a5a
b638e5 /*
2af37a /* PC2: Map 56-bit round key to a 48-bit subkey
d5495d */
o97fcf static int table DES_PC2[48] = {
  e9b889 ...... 24, 27, 20, -6, 14, 10, -3, 22,
  1b3a5 ...... 0, 17, -7, 12, -8, 23, 11, -5,
  893fa5 ...... 16, 26, -1, -9, 19, 25, -4, 15,
  dee272 ...... 54, 43, 36, 29, 49, 40, 48, 30,
  566556 ...... 52, 44, 37, 33, 46, 35, 50, 41,
  ab7786 ...... 28, 53, 51, 32, 43, 39, 42
2a82f7);
11a5a
c0a5a
2238e5 /*
87feb1 /* E: Expand 32-bit R to 48 bits.
35495d */

86a807 static int table DES_E[48] = {
  7beb6b ...... 31, -0, -1, -2, -3, -4, -3, -4,
  fa2634 ...... 5, -6, -7, -8, -7, -8, -9, 10,
  1dbd6b ...... 11, 12, 11, 12, 13, 14, 15, 16,
  1d787b ...... 15, 16, 17, 18, 19, 20, 19, 20,
  184d0d ...... 21, 22, 23, 24, 25, 24, 25, 26,
  139708 ...... 27, 28, 27, 28, 29, 30, 31, -0
8a82f7);
79a5a
c0af5a
cd5a5a
a738e5 /*
6c34a /* P: Permutation of S table outputs
ea495d */

7d5137 static int table DES_S[32] = {
  aaf612 ...... 11, 17, -5, 27, 25, 10, 20, -6,
  3f9bf8 ...... 13, 21, -3, 28, 29, -7, 18, 24,
  8f2e3 ...... 31, 22, -12, -6, 26, -2, 16, -8,
  d568a7 ...... 14, 38, -4, 19, -1, 9, 15, 23
8482f7);
79a5a
8ea5a
d038e5 /*
2b6f34 /* S Tables: Introduce nonlinearity and avalanche
d2495d */
b7fe9 static int table DES_S[56] = {
  86b89a ...... */ table SE[j] */
e1846d ........ ( 13, -1, -2, 15, -8, 13, -4, -8, -6, 10, 15, -3, 11, -7, 1, -4,
--if22 000be80f13a804007 Page 3 of des.c

f05ef 10, 12, -9, -5, 3, 6, 14, 11, -5, 0, 0, 14, 12, 9, 7, 2, 12, 5, 0, 0, 0, 0, 14, 11, 7, 1, 9, 4, 12, 10, 14, 8, -2, 13, 0, 5, 3, 3, 5, 5, 6, -8, 11, ...

d02b1 /* table $C[1]/

8b66a /* table $C[2]/

8c86d /* table $C[3]/

8bf575 /* table $C[4]/

cf234 /* table $C[5]/

38dec /* table $C[6]/

7df108 /* table $C[7]/

a3d828 /* table $C[8]/

def5d /* table $C[9]/

137585 /* table $C[10]/

9fda7 /* table $C[11]/

f354c4 /* table $C[12]/

74850 /* table $C[13]/

e202b /* table $C[14]/

35bd2c /* table $C[15]/

55d567 /* table $C[16]/

9b26b /* table $C[17]/

0e76b /* table $C[18]/

f8c66 /* table $C[19]/

baac1c /* table $C[20]/

67731 /* table $C[21]/

f8b2a /* table $C[22]/

1c1a4b /* table $C[23]/

0d82f7

4a5f9

8a5f9

1eaf5a

be4d0c

ec8f /* DES CODE */

77c6bf /* DES CODE */

e4c9d0c /* DES CODE */

eaf5f9

91385 /* EncryptDES: Encrypt a block using DES. Set verbose for debugging info. */

1b050 /* (This loop does both loops on the "DES Encryption" page of the flowchart.) */

e14d95 /*

5a5620 void EncryptDES(block key[56], block outBlk[64], block inBlk[64], int verbose) {

d72b1c // int i, Round;

3a9a1 // block RC[32], L[32], fout[32];

1cbf9 // block roundKey[56];

24a5f9

eb942 // EnableDumpBin = verbose; /* set debugging on/off flag */

3b2b2a // DumpBin("input(left)", inBlk[32], 32);

558fb2 // DumpBin("input(right)", inBlk, 32);

c0be8 // DumpBin("raw key(left)", key+28, 28);

8a585 // DumpBin("raw key(right)", key, 28);

eaf5f9

f4c1be // Compute the first roundkey by performing PC1 */

53b26 // ComputeRoundKey(roundKey, key);
Chapter 7: Chip Simulator Source Code

---

d8d1a8  ComputeIP(L,R,inBlk);

c8a5a  DumpBin("after IP(L)", L, 32);

e47699  DumpBin("after IP(R)", R, 32);

56a5a  if (verbose)

4af47  for (round = 0; round < 16; round++) {

d221bf  if (verbose)

5f91a0  printf("------------ BEGIN ENCRYPT ROUND %d ---------------\n", round);

aa8034  DumpBin("round start(L)", L, 32);

ea8117  DumpBin("round start(R)", R, 32);

d8a5a  f27fc3  /* Rotate roundKey halves left once or twice (depending on round) */

3ec8ba  RotateRoundKeyLeft(roundKey);

03f467  if (round != 0 && round != 1 && round != 8 && round != 15)

9650e7  RotateRoundKeyLeft(roundKey);

dd3cd7  DumpBin("round(L)", roundKey+28, 28);

3f1bd4  DumpBin("round(R)", roundKey, 28);

8ba5f  ca033b  /* Compute f(R, roundKey) and exclusive-OR onto the value in L */

73d969  ComputeF(fout, R, roundKey);

1054e7  DumpBin("f(R, key)", fout, 32);

8f4739  for (i = 0; i < 32; i++)

ea9e66  L[i] ^= fout[i];

aca5ab  DumpBin("L^f(R, key)", L, 32);

92af5a  //f08e88  Exchange_L_and_R(L,R);

c2a5a  a51a40  DumpBin("round end(L)", L, 32);

9a0d63  DumpBin("round end(R)", R, 32);

0c21bf  if (verbose)

ee4514  printf("------------- END ROUND %d -------------\n", round);

bed1fc  

39a5a  f08e68  Exchange_L_and_R(L,R);

7ba5a  a7370b  /* Combine L and R then compute the final permutation */

a3c9f4  ComputeFP(outBlk,L,R);

668b91  DumpBin("FP out(L)", outBlk+32, 32);

af6f75  DumpBin("FP out(right)", outBlk, 32);

bba5f  f0aa4  Combine L and R then compute the final permutation */

52b9a  DumpBin("L[C56], L[32], fout[32];

52b9a  boolean R[32], L[32], fout[32];

7ba5a  a7370b  void DecryptDES(boolean key[56], boolean outBlk[64], boolean inBlk[64], int verbose) {

7e495d  /*

c36de8  DecryptDES (boolean key[56], boolean outBlk[64], boolean inBlk[64], int verbose) {

cb21c  int i, round;

1a99a1  boolean R[32], L[32], fout[32];

52b9a  boolean roundKey[56];

87a5f  a9e9e2  EnableDumpBin = verbose; ................................./* set debugging on/off flag */

6e99f2  DumpBin("input(left)", inBlk+32, 32);

34a08e  DumpBin("raw key(left)", key+28, 28);

ed5585  DumpBin("raw key(right)", key, 28);

90a5f  3ac1be  /* Compute the first round key by performing PC1 */

20b264  ComputeRoundKey(roundKey, key);

a5a5a  c595d4  DumpBin("roundKey(L)", roundKey+28, 28);

c9a05f  DumpBin("roundKey(R)", roundKey, 28);

f4af5a  311340  /* Compute the initial permutation and divide the result into L and R */

03d1a8  ComputeIP(L,R,inBlk);

a8a5a  c5a77b  DumpBin("after IP(L)", L, 32);

2f7699  DumpBin("after IP(R)", R, 32);

a5a5a  1bf437  for (round = 0; round < 16; round++) {

4f21bf  if (verbose)

---

Page 4 of des.c
Chapter 7: Chip Simulator Source Code

```
--4cd6 000642b0cd180040007 Page 5 of des.c
db2cb4...printf("---------- BEGIN DECYPRT ROUND %d----------
", round);
980834...DumpBin("round start(L)", L, 32);
a84117...DumpBin("round start(R)", R, 32);
c5a5a...// Compute f(R, roundKey) and exclusive-OR onto the value in L */
b5d969...ComputeF(fout, R, roundKey);
5d5ae7...DumpBin("f(R, key)", fout, 32);
c84739...for (i = 0; i < 32; i++)
d5a9e6...L[i] = fout[i];
5ba5ab...DumpBin("L^f(R, key)", L, 32);
b2a5a...fa68b4...Exchange_Land_R(L, R);
18a5a...f3c90d.../* Rotate roundKey halves right once or twice (depending on round) */
6e4bf3...DumpBin("roundKey(L)", roundKey+28, 28); /* show keys before shift */
7f1bd4...DumpBin("roundKey(R)", roundKey, 28);
90f5db...RotateRoundKeyRight(roundKey);
871ff...if (round != 0 && round != 7 && round != 14 && round != 15)
ba7c23...RotateRoundKeyRight(roundKey);
5ba5a...69a140...DumpBin("round end(L)", L, 32);
c0a663...DumpBin("round end(R)", R, 32);
f621bf...if (verbose)
784514...printf("---------- END ROUND %d----------
", round);
23df1c...)
30a5a...ec8e68...Exchange_Land_R(L, R);
f5a5a...f5c94...ComputeFP(outBlk, L, R);
8d8b91...DumpBin("out Blk", outBlk+52, 32);
94f675...DumpBin("out (right)", outBlk, 32);
eff66...}
3da5a...deaf5a...41a5a...
f38e5/*
4b8dbb...* ComputeRoundKey: Compute PC1 on the key and store the result in roundKey
1c495d...*/
3e988e...static void ComputeRoundKey(bool roundKey[56], bool key[56]) {
9e17e0...int i;
60a5a...f9815b...for (i = 0; i < 56; i++)
70d6db...roundKey[table DES PC1[i]] = key[i];
54ef6...}
24af5a...8caf5a...daaf5a...
1b38e5/*
8155cb...* RotateRoundKeyLeft: Rotate each of the halves of roundKey left one bit
44495d...*/
737d60...static void RotateRoundKeyLeft(bool roundKey[56]) {
cf483e...-bool temp1, temp2;
727ef0...int i;
0f6f89...temp1 = roundKey[27];
83fe1b...temp2 = roundKey[55];
9c308b...for (i = 27; i >= 1; i--) {
95575a...roundKey[i] = roundKey[i-1];
0b3242...roundKey[i+28] = roundKey[i+28-1];
aa9f1c...}
bc7bbf...roundKey[0] = temp1;
b3c19d...roundKey[28] = temp2;
51fe6...}
fbaf5a...5ca5a...51a5a...
0c38e5/*
3b6c...* RotateRoundKeyRight: Rotate each of the halves of roundKey right one bit
87495d...*/
57b26c...static void RotateRoundKeyRight(bool roundKey[56]) {
64483e...-bool temp1, temp2;
```
Chapter 7: Chip Simulator Source Code

Chapter 7: Chip Simulator Source Code

--20a0 0000e89a59d480040007 Page 6 of des.c

```
--20a0 0000e89a59d480040007 Page 6 of des.c

eb17e0  int i;
71a5fa 085025  temp1 = roundKey[0];
7545f8  temp2 = roundKey[28];
78e568  for (i = 0; i < 27; i++) {
800cc2  roundKey[i] = roundKey[i+1];
fd09da  roundKey[i+28] = roundKey[i+28+1];
c1df1c  }
d5a88d  roundKey[27] = temp1;
365d11  roundKey[53] = temp2;
29e6e6  }
5da5fa 4a5fa5  d8a5fa 2f38e5 /*
022983  * ComputeIP: Compute the initial permutation and split into L and R halves.
fb695d  static void ComputeIP(bool L[32], bool R[32], bool inBlk[64]) {
826085  bool output[64];
917e0  int i;
bdaf5a 81aeaf  /* Permute */
71f9a6  466406  for (i = 63; i >= 0; i--)
ac7c50  outputTable[DES.IP][i] = inBlk[i];
0aaf5a  af0318  /* Split into R and L. Bits 63..32 go in L, bits 31..0 go in R.
54f9a6  */
b8aa85  for (i = 63; i >= 0; i--){
5b9368  if (i >= 32)
67f2b8  L[i-32] = output[i];
5084c  else
7970b5  R[i] = output[i];
22df1c  }
04e6e6  }
46a5fa c0af5a 96a5fa 6a38e5 /*
03c6e1  * ComputeFP: Combine the L and R halves and do the final permutation.
56495d  static void ComputeFP(bool outBlk[64], bool L[32], bool R[32]) {
ee42e9  bool input[64];
0b17e0  int i;
0fa5a 56c61  /* Combine L and R into input[64]
30f9a6  */
836406  for (i = 63; i >= 0; i--)
5a8397  input[i] = (i >= 32) ? L[i - 32] : R[i];
4caf5a 97aeaf  /* Permute */
1cf9a6  016486  for (i = 63; i >= 0; i--)
fe8116  outBlkTable[DES.FP][i] = input[i];
c4e6e6  }
3baf5a 13af5a  b9a5fa 4a38e5 /*
33810f  * ComputeF: Compute the DES f function and store the result in fout.
144f9d  */
a272b0  static void ComputeF(bool fout[32], bool R[32], bool roundKey[56]) {
97f6a2  bool expandedBlock[48], subkey[48], sout[32];
51b6e6  int i, k;
23af5a 291a04  /* Expand R into 48 bits using the E expansion */
a599d7  ComputeExpansionE(expandedBlock, R);
f1f0ba  DumpBin("expanded E", expandedBlock, 48);
81af5a 7c93ff  /* Convert the roundKey into the subkey using PC2 */
17840  ComputePC2(subkey, roundKey);
e8d717  DumpBin("subkey", subkey, 48);
```

Chapter 7: Chip Simulator Source Code

---4509 0001cf13c2680040007  Page 7 of des.c

e0af5a  c3154c  ../* XOR the subkey onto the expanded block */
adfcab  ..for (i = 0; i < 48; i++)
4f6512  ..expandedBlock[i] ^= subkey[i];
60af5a  b87040  ../* Divide expandedBlock into 6-bit chunks and do S table lookups */
1d25c6  ..for (k = 0; k < 6; k++)
6585c7  ..ComputeS_Lookup(k, sout+4*k, expandedBlock+6*k);
8f8af5a  b3fd35  ..*/ To complete the f() calculation, do permutation P on the S table output */
e92df2  ..ComputeP(fout, sout);
3ba5fa  }  3ba5fa
25af5a  0af5a
8438e5  /*
600913f  ..ComputeP: Compute the P permutation on the S table outputs.
25495d  */
0b4610  static void ComputeP(bool output[32], bool input[32]) {
5a17e0  ..int i;
b6af5a  95339a  ..for (i = 0; i < 32; i++)
347688  ..output[table-DES_P[i]] = input[i];
57efe6  }
3ba5fa  80af5a
67af5a  75af5a
e638e5  */
2a859b  ..Look up a 6-bit input in S table k and store the result as a 4-bit output.
2e495d  */
59af6e  static void ComputeS_Lookup(int k, bool output[4], bool input[6]) {
d9f3da  ..int inputValue, outputValue;
3da5fa  261a9e  ..*/ Convert the input bits into an integer */
468c1a  16*input[4] + 32*input[5];
1caf5a  a64a3e  ..*/ Do the S table lookup */
a2b706  ..outputValue = table-DES_S[k][inputValue];
eea5fa  dca5fa  ..*/ Convert the result into binary form */
4f6af5a  529a60  ..output[0] = (outputValue & 1) ? 1 : 0;
2c6a0a  ..output[1] = (outputValue & 2) ? 1 : 0;
a4f687  ..output[2] = (outputValue & 4) ? 1 : 0;
288cf7  ..output[3] = (outputValue & 8) ? 1 : 0;
27efe6  }
dcaf5a  df5af5a
f4af5a  5938e5  /*
9781cc  ..*/ ComputePC2: Map a 56-bit round key onto a 48-bit subkey
a8495d  */
07796f  static void ComputePC2(bool subkey[48], bool roundKey[56]) {
e17e0  ..int i;
2fa5fa  64fcab  ..for (i = 0; i < 48; i++)
c3c8bc  ..subkey[i] = roundKey[table-DES_PC2[i]];
f8ef6  }
89af5a  4eaf5a
8baf5a  3b38e5  */
7a459d  ..*/ ComputeExpansionE: Compute the E expansion to prepare to use S tables.
89495d  */
89b66d  static void ComputeExpansionE(bool expandedBlock[48], bool R[32]) {
bf17e0  ..int i;
94af5a  79fcab  ..for (i = 0; i < 48; i++)
d9bf71  ..expandedBlock[i] = R[table-DES_E[i]];
8efe6  }
9daf5a  1caf5a
--bb13 0016ae66eb080040007 Page 8 of des.c

e0af5a
d38e55 /*
4cf923  */ Exchange_L_and_R: Swap L and R
b495d */
f195d1 static void Exchange_L_and_R(bool LE32, bool RE32) {
  3f17e0  ...int i;
  c2af5a  for (i = 0; i < 32; i++)
  19fe8b  ...LE[i] ^= RE[i]; .../* exchanges LE[i] and RE[i] */
  3cefe6  }
  4af5a  }
72af5a  feaf5a
a438e55 /* DumpBin: Display intermediate values if enableDumpBin is set.
52a495d */
9cdbd9 static void DumpBin(char *str, bool *b, int bits) {
  617e0  ...int i;
  80af5a  ...if ((bits % 4)!=0 || bits>48) {
  c98af7  ....printf("Bad call to DumpBin (bits > 48 or bit len not a multiple of 4\n");
  17b2e5  ....exit(1);
  4b46c  ....}
  adaf5a  }
783322 ...if (EnableDumpBin()) {
  351079  ...for (i = strlen(str); i < 14; i++)
  1ac8c3  ....printf(" ");
  1c5f73  ....printf("%s: ", str);
  56eac8  ...for (i = bits-1; i >= 0; i--)
  2d0e5b  ....printf("%d", b[i]);
  123177  ....printf(" ");
  9c821f  ...for (i = bits; i < 48; i++)
  72c8c3  ....printf(" ");
  d86b57  ....printf(" ");
  9105d7  ...for (i = bits-4; i >= 0; i-=4)
  c6178c  ....printf("%X", b[i]+2*b[i+1]+4*b[i+2]+8*b[i+3]);
  89af6f  ....printf("\n");
  e4df1c  }
56eaf6  }
6aaf5a
```
typedef char bool;

void EncryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);

void DecryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);
```
Chapter 7: Chip Simulator Source Code

7-17

--842b 001c97afa9d80040009 Page 1 of ecb.scr

f64ce1 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A
fdb9bc A1 10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A
b5f9a7 30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A
4c9b42 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65
5b18b1 66 67 68 69 6A 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
5c23c9 80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A
4a1c73 9B 9C 9D 9E 9F A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF B0 B1 B2 B3 B4 B5 B6
36fc9a B7 B8 B9 BA BB BC BD BE BF C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF D0 D1 D2
5b5b61 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF EF E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED
ee9439 EF FF FF FF FF 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17
829ec4 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33
b74a75 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
642ba4 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A
65c198 6B 6C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F 80 81 82 83 84 85
aacad1 86 87 88 89 8A 8B 8C 8D 8E 8F 90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
ff84f0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
65c198 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15 16 17 18 19 1A
332fcc 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 34 35
769df9 36 37 38 39 3A 3B 3C 3D 3E 3F 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50
14af5a 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F 60 61 62 63 64 65 66 67 68 69 6A

XOR MASK
use CBC
extra XOR
don't seed PRNG (use this input file)
starting key
number of clocks

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include "des.h"
#include "misc.h"

#define VERBOSE

1ac562 void getUserInfo(unsigned char plaintextVector[32],
7c3170 while (1--
5f4b72a int hex2bin(char *hex, unsigned char *data, int len);
7e11f4 void printHexString(char *hex, unsigned char *tag, unsigned char *data, int len);
9a2800748214c28004000b Page 1 of misc.c

Chapter 7: Chip Simulator Source Code

7-19
--9cbe 0004b9f4b0980040000 Page 2 of misc.c

74766e .get(buffer);
82b72a .i = hex2bin(buffer, tmp);
886486 if (i != 1)
b7f706 "EXIT-ERR( "Plaintext byte mask is 1 byte long.");
6bc2b5 *plaintextByteMask = tmp[0];

7a75a #ifndef VERBOSE
7d1b21 printf(" ......Enter useCBC (0 or 1): ");

f57a54 #endif

56766e .get(buffer);
eab72a .i = hex2bin(buffer, tmp);
e515b9 .if (i != 1 || tmp[0] > 1)
b819c6 "EXIT-ERR( "Must enter 0 or 1 for useCBC.");
6be2c1 *useCBC = tmp[0];

37a5a 264264 #ifdef VERBOSE
7bf965 printf(" ......Enter extraXor (0 or 1): ");

887454 #endif

4e766e .get(buffer);
61b72a .i = hex2bin(buffer, tmp);
7e15b9 .if (i != 1 || tmp[0] > 1)
77bc75 "EXIT-ERR( "Must enter 0 or 1 for extraXor.");
522353 *extraXor = tmp[0];

29a5a

a04264 #ifdef VERBOSE
a7e8e1 printf(" ......Enter quickStart (0 or 1): ");

757454 #endif

c766e .get(buffer);
19b72a .i = hex2bin(buffer, tmp);
a015b9 .if (i != 1 || tmp[0] > 1)
b12169 "EXIT-ERR( "Must enter 0 or 1 for quickStart
");
83bd7b *quickStart = tmp[0];

91a5a

0eaf5a #ifdef VERBOSE
02c8bf .printf(" ........Enter starting key: ");

187454 #endif

d3766e .get(buffer);
0e8e84 .if (hex2bin(buffer, tmp) != 7)
f6a5e4 "EXIT-ERR( "Must enter 7 hex bytes as the key.
");
130ed3 .memcpy(startKey, tmp, 7);

80a5a

514264 #ifdef VERBOSE
434f66 .printf(" ......Enter number of clocks: ");

1c7454 #endif

c677f7(method, "%ld", numClocks);
4bb8b1 .if (*numClocks < 1 || *numClocks > 100000000L)
d6b893 "EXIT-ERR("Must have between 1 and 1 billion clocks.
");
6da5a

134264 #ifdef VERBOSE

dc186d .printf("....PlaintextVector = ", plaintextVector, 32);
a4a738 .printf("..PtxtXorMask = ", PtxtXorMask, 8);
5dfc6 .printf("Plaintext = ", plaintext, 8);
9b7b6 .printf("...Ciphertext = ", ciphertext, 1);
99dd31 .printf("...Ciphertext = ", ciphertext, 1);
72e15b .printf(".....useCBC = %d", *useCBC);
2b2f3b .printf("......extraXor = %d", *extraXor);
68b72c .printf(".........quickStart = %d", *quickStart);
5f11535 .printf("...Total clocks = %d", *numClocks);

517454 #endif

ebf7e0 )

7ca5a

25af5a void increment32(unsigned char *v) {

fae2f8 .if (++(v[3]) == 0)
77ab75 .if (++(v[2]) == 0)
74b31a .if (++(v[1]) == 0)
584858 .++v[0];
9bfef6 )

4ba5a
void decrement32(unsigned char *v) {
    if (((v[3]) == 0))
        v[2] = 0;
    else
}

void desDecrypt(unsigned char m[8], unsigned char c[8], unsigned char k[7]) {
    bool key[56], message[64];
    fd17e0 = int i;
    68af5a
    54ec0c // print("DES-DECRYPT(k="," for (i=0; i<7; i++) printf("%02X",k[i]); //!!!
    4cf314 // print("", c="); for (i=0; i<8; i++) printf("%02X",c[i]); //!!!

    if (k[i] == 1)
        for (i=0; i<64; i++)
            if (k[i] != k[i+1])
                continue;

    85af4b // DecryptDES(key, message, message, 0);
    5bf176 // for (i = 0; i < 8; i++)
    6f452e // m[i] = 0;
    f55333 // for (i = 0; i < 64; i++)
    fa7511 // if (message[i] == 1)
    59f2e9 // for (i = 0; i < 8; i++)
    8a41e6 // printf("%02X",m[i]); printf("\n"); //!!!
    b8af5a
    43ef6e )
    17aaf5a
    63a5f5a
    30c2bf int unhex(char c) {
        bc53c4 // if (c >= '0' && c <= '9')
        1683d6 // return (c - '0');
        8d8bb1 // if (c >= 'a' && c <= 'f')
        480ada // return (c - 'a' + 10);
        3449e3 // if (c >= 'A' && c <= 'F')
        28a66f // return (c - 'A' + 10);
        c712d4 // return (-1);
    }
    2eef6e )
    85af5a
    46a5f5a
    64a5f5a
    64a5f5a
    59be6f /* Trim string if comments present */
    b08b28 // if (strchr(hex, '\n') != NULL)
    c252e6 // *strchr(hex, '\n') = 0;
    1ba3ed // if (strchr(hex, '*') != NULL)
    7761d0 // *strchr(hex, '*') = 0;
    e7a6e5 // if (strchr(hex, '\') != NULL)
    7c2b42 // *strchr(hex, '\') = 0;
    aabf5a
    5a1f4a // for (i = 0; i < strlen(hex); i++)
    6a8e69 // if (hex[i] >= '0' && hex[i] < 0)
    fed278 // EXIT_ERR("Bad hex digit encountered.
    09df1c )
    4baf5a
    41f14a // for (i = 0; i < strlen(hex); i++)
    c1f5b0 // if (hex[i] < 'a')
    2b56e4 // continue;
    266643e // if (hex[i] == '0' && hex[i+1] >= '0')
    567935 // *bin[i++] = unhex(hex[i])*16+unhex(hex[i+1]);
    db028 // i++; // skip one
    195f6a // continue;
    8b66fe //
    3f16f // if (hex[i] == '0')
    a78539 // *bin[i++] = unhex(hex[i]);
    aa6ef7 //
    edf13c //
    68c1d2 // return (j);
    5def6e }
```c
void printHexString(char *tag, unsigned char *data, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%02X", data[i]);
    printf("\n");
}
```
Chapter 7: Chip Simulator Source Code

---77c4 001029468fd8004000c Page 1 of misc.h

e0af5a
32c502 void GetUserInfo(unsigned char plaintextVector[32],
6657e7 ........unsigned char plaintextXorMask[8],
e5910f ..........unsigned char ciphertext0[8], unsigned char ciphertext[8],
4c446e ........unsigned char *plaintextByteMask, int *useCBC, int *extraXor,
25e00d ........int *quickStart, unsigned char startKey[7], Long *numClocks);
560986 void increment32(unsigned char *v);
edb70b void decrement32(unsigned char *v);
4cf314 void desDecrypt(unsigned char m[8], unsigned char c[8], unsigned char k[7]);
fea5c5 void printHexString(char *tag, unsigned char *data, int len);
f9560a int hex2bin(char *hex, unsigned char *bin);
02af5a
--91c4 001d95d620a8004000d Page 1 of random.scr

1b9f56 00
4e9ec4 0000000000000000 ...' XOR MASK
c4327c 0000000000000000 ...' Ciphertext 0
892ba4 0000000000000000 ...' Ciphertext 1
918c19 00 .........................' Plaintext byte mask
05ed5e 0 .........................' use CBC
37d84f 0 .........................' extra XOR
260627 0 .........................' random vector (0=seed with timer, 1=use input, >1=seed)
63a481 0000000000000000 .........' starting key
8bd03f 2000 .........................' number of clocks
d9af5a
Chapter 7: Chip Simulator Source Code

7-24

--0289 000317f111e8004000 Page 1 of ref.c

56feb2 #include <stdio.h>
a1bea3 #include <stdlib.h>
a9c737 #include <memory.h>
79324c #include <string.h>
24bac #include "des.h"
3ada5a #include "misc.h"
feaf5a
e7f8b5 #define CLOCKS_PER_DES 18
d8af5a
7a4525 int plaintextMatch(unsigned char plaintextVector[32], unsigned char m[8],
c46e85 --------unsigned char plaintextByteMask, int ciphertext, unsigned char key[7]);
8787d4 void checkKey(unsigned char key[7], unsigned char plaintextVector[32],
4357e7 --------unsigned char plaintextXorMask[8],
d37f8 --------unsigned char ciphertext0[8], unsigned char ciphertext1[8],
7d5e1b --------unsigned char plaintextByteMask, int useCBC, int extraXor);
64af5a
57a2c4 void main(void) {
2b7ab8 unsigned char startKey[7], plaintextVector[32];
776f32 --------unsigned char plaintextXorMask[8];
01ed17 --------unsigned char ciphertext0[8];
52e93c --------unsigned char ciphertext1[8];
31ff49 --------unsigned char plaintextByteMask;
36cd98 --------int useCBC, extraXor, quickStart;
38e13a --------i, j; 948520 --------long numClocks;
ac5ec8 --------unsigned char key[7];
d7af5a
8a5e03 GetUserInfo(plaintextVector, plaintextXorMask, ciphertext0, ciphertext1,
87df56 --------&plaintextByteMask, &useCBC, &extraXor, &quickStart, startKey,
47ab5c --------&numClocks); 2aa5f5a
7fe44 --------for (i = 0; i < numClocks; i += CLOCKS_PER_DES) {
dea2d3 --------for (j = 0; j < 24; j++) {
22b4be --------memcpy(key, startKey, 8);
08c578 --------key[0] = j;
ab91ec --------checkKey(key, plaintextVector, plaintextXorMask, ciphertext0,
5320a7 --------ciphertext1, plaintextByteMask, useCBC, extraXor); 3e6f87 --------)
71ddad --------increment32(startKey+3);
96df1c --------} aee6f6 }
8ca5a
08f5d5a
d287d4 void checkKey(unsigned char key[7], unsigned char plaintextVector[32],
5c5e7e --------unsigned char plaintextXorMask[8],
38e9f8 --------unsigned char ciphertext0[8], unsigned char ciphertext1[8],
544479 --------unsigned char plaintextByteMask, int useCBC, int extraXor) {
3c0a0b --------unsigned char m[8];
1417e0 --------i; d4af5a
a4f09d --------desDecrypt(m, ciphertext0, key);
26b35d --------printf("DEScrypt(km)\r\n"), for (i = 0; i < 7; i++) printf("%02X", key[i]);
188cf0 --------printf("\r\n", C0="), for (i = 0; i < 8; i++) printf("%02X", ciphertext0[i]);
d484c7 --------printf("\r\n", 24667c --------if (extraXor) {
0bd985 --------m[0] ^= m[4];
db40eb --------m[1] ^ m[5];
f2e248 --------m[2] ^= m[6];
607ba6 --------m[3] ^= m[7];
7cdflc --------}
bb1f76 --------for (i = 0; i < 8; i++)
48ee88c --------m[3] ^= plaintextXorMask[i];
17f157 c64534 --------if (plaintextMatch(plaintextVector, m, plaintextByteMask, 0, key)) {
5b349f --------desDecrypt(m, ciphertext1, key);
96553a --------printf("DEScrypt(km)\r\n"), for (i = 0; i < 7; i++) printf("%02X", key[i]);
6dd4d --------printf("\r\n", C1="), for (i = 0; i < 8; i++) printf("%02X", ciphertext1[i]);
eadb21 --------printf("\r\n"), for (i = 0; i < 8; i++) printf("%02X", m[i]);
37952a --------if (extraXor) {
af2ad3 --------m[0] ^= m[4];
bb6b3d --------m[1] ^= m[5];
31111e --------m[2] ^= m[6];

Chapter 7: Chip Simulator Source Code

--62a0 0018ac4c1498004000e Page 2 of ref.c

4c8f0 ...... m[3] ^= m[7];
5b6fe7 ......
5b662a ...... if (useCBC) {
41f494 ...... for (i = 0; i < 8; i++)
e13f39 ...... m[i3] ^= ciphertext[13];
2b6fe7 ......
10e5be ...... if (plaintextMatch(plaintextVector, m, plaintextByteMask, 1, key)) {
5246ff ...... printf("--- VALID MATCH-------\n");
059a8a ...... fprintf(stderr, "Match found at key =");
c2526 ...... for (i = 0; i < 7; i++)
3f44d0 ...... fprintf(stderr, "%02X", key[i]);
0f5501 ...... fprintf(stderr, "\n");
726fe7 ......
6ad1f1c ...
19e6e6 )
81af5a
81af5a
004525 int plaintextMatch(unsigned char plaintextVector[32], unsigned char m[8],
8aa762 ...... unsigned char plaintextByteMask, int ciphertext, unsigned char key[7]) {
8017f0 ...... int i;
6ff65a
068715 ...... for (i = 0; i < 8; i++) {
5c58ef ...... if ((plaintextByteMask & (128>>i)) == 1)
388e03 ...... continue; /* this byte is skipped */
b98ec8 ...... if (plaintextVector[m][i] & (128 >> (m[1][8])))
d65f6a ...... continue;
854210 ...... return (0); ........../* no match */
13df1c ...
03af5a
957381 ...... printf("Match of C%d with key ", ciphertext);
2dace4 ...... for (i = 0; i < 7; i++)
800dfc ...... printf("%02X", key[i]);
b461e2 ...... printf(" = ");
021f76 ...... for (i = 0; i < 8; i++)
65f1eb ...... printf("%02X", m[i]);
b7f6eb ...... printf("\n");
93af5a
35ec77 ...... fprintf(stderr, "Match of C%d with key ", ciphertext);
1dace4 ...... for (i = 0; i < 7; i++)
5feda0 ...... fprintf(stderr, "%02X", key[i]);
d29864 ...... fprintf(stderr, " = ");
371f76 ...... for (i = 0; i < 8; i++)
d7651e ...... fprintf(stderr, "%02X", m[i]);
90c77e ...... fprintf(stderr, "\n");
10af5a
e041d1 ...... return (1);
9af6e6 )
81af5a
8aa5af
--c93c 000f4b51c08004000f Page 1 of sim.c

Chapter 7: Chip Simulator Source Code

7-27
--- ba12 0009cc27d6a8004000f Page 2 of sim.c
417126 /* DES_POSITION = 13; */
9fa8d9 /* memset(WORKING_CTX0, 0, sizeof(WORKING_CTX0)); */
e2ae08 /* memset(WORKING_PTXT, 0, sizeof(WORKING_PTXT)); */
b5af0b /* memset(RAW_DES_OUT, 0, sizeof(RAW_DES_OUT)); */
6f5ba0 /* memset(WORKING_KDELTA, 0, sizeof(WORKING_KDELTA)); */
c1e6c6 /* memset(WORKING_LAST_SELECTOR, 1, sizeof(WORKING_LAST_SELECTOR)); */
c48102 /* memset(WORKING_NEXT_SELECTOR, 1, sizeof(WORKING_NEXT_SELECTOR)); */
e990bd /* memset(STARTUP_DELAY, 0, sizeof(STARTUP_DELAY)); */
8f3f0e /* memset(this_KEY, 0, sizeof(this_KEY)); */
849161 /* memset(NEXT_KEY, 0, sizeof(NEXT_KEY)); */
850c35 /* PENDING_UPDATE_ADDR1 = -1; */
c98635 /* PENDING_UPDATE_ADDR2 = -1; */
2ba08a /* PENDING_UPDATE_ADDR3 = -1; */
41f9ae /* memset(MATCH, 0, sizeof(MATCH)); */
49f4e6 /* */
92af9a /* */
94af9a /* */
51a5b0 long getClockCounter(void) {
  8eb11d return (CLOCK_COUNTER);
  7efef6 /* */
  85a5f5 /* */
  98af9a /* */
  5f22bd int peekState(int addr) {
    c69e96 return (STATE[addr]);
  78ef60 /* */
  c0af5a /* */
  7caf5a /* */
  6686c4 int RunChip(char *input, FILE *outfile, int useRaw) {
  6d8717 /* */
  129d7f /* */
  29b7df /* */
  4e7f49 /* ALLACTIVE_IN = (unsigned char)allactiveIn; */
  38a5f9 /* */
  183f23 /* dataOut = data; .......... ... default */
  9d58d6 /* if (reset == 0) ( ............... /* reset? */
  54b64f /* */
  a2b1d3 /* */
  129d7f /* */
  3bcb60 /* */
  bb80de /* SELECTED_CHIP = (unsigned char)chipId; */
  9884c2 /* */
  21929b /* */
  7302c9 /* */
  8781d3 /* */
  64d581 /* */
  df81d3 /* */
  ebb109 /* */
  75b9c0 /* */
  3bda60 /* */
  b46b08 /* */
  c78591 /* */
  f805d7 /* */
  c78591 /* */
  beff30 /* */
  9c6fe7 /* */
---eb12 000b9c2388b800000f Page 3 of sim.c

5be4b8 .. else if (rdb == 0) { ................. /* read a register */
e3a499 ....dataOut = STATE[addr];
2e81d3 ....RunClock();
1249d8 ..} else {
0981d3 ....RunClock();
فاد� ....}
eaa5fa
671c5c ..if (CLOCK_COUNTER >= 2) {
123ce ...} if (useRaw) {
e1a3f4 ....fprintf(outfile, "%02X %d\n", dataOut, ALLACTIVE_OUT);
9c6a79 ..} else {
fe2044 ....fprintf(outfile, " (Addr: %02X) (Exp: 00) (Get: %02X) at Cycle:%d\n",
e9f86f ...........addr, dataOut, CLOCK_COUNTER);
01457c .......for (i = 0; i < 24; i++) {
31e69a ........for (j = 6; j >= 0; j--)
ff59a .........fprintf(outfile, "\n",
97f7e5 .........STATE[REG_SEARCH_KEY(i)+j]);
3555b .........if (CLOCK_COUNTER < 22)
78dc93 ........fprintf(outfile, "0000000000000000\n");
ff0e0 ........else if (CLOCK_COUNTER <= 37)
3054a .........fprintf(outfile, "094CE83D677160F\n");
830d7 .........else {
78f9b0 ........for (j = 7; j >= 0; j--)
5ca2d2 ........fprintf(outfile, "%02X", RAW_DES_OUT[8*i+j]);
37f1f0 }
9978b2 #if 0 ............................. /* uncomment to print information about the MATCH */
42b60c ........static int latch[24]=
451fe8 ...} static int latch[24]=
4555b3 ...} static int latch[24]=
2c123c ...} if (DES_POSITION==10) [ latch[i] = MATCH[i];
a6a8b3 ........fprintf(outfile, "%d", latch[i]);
7f7bf0 }
017454 #endif
375a5 ........fprintf(outfile, " : Unit%\n", i);
6a2cc2 
2b96e ...fprintf(outfile, "\n");
e86ef7 ...
23df1c ...
a3755d ..CLOCK_COUNTER++;
075993 ..return (dataOut);
0CEF66 }
0CE3b5a 
e7a5f5a 
d5a70e static void parseInput(char *input, int *reset, int *boardEn, int *ale,
ce8541 ...} int *adrsel1, int *web, int *rdb, int *adrsel2, int *allactIn,
76e4a5 ...} int *boardEn, int *chipId, int *data) {
6a7e0 ...} int i;
47a5f5a ...
c0a5f ...} EXIT_ERR("Bad input\n");
558715 ...for (i = 0; i < 8; i++) {
c1c317 ...} if (input[i] != '0' & & input[i] != '1')
93b045 ...} EXIT_ERR("Bad input (first 8 digits must be binary.)\n");
c1d4fc ...
3b95f8 ..} if (unhex(input[9]) < 0 || unhex(input[10]) < 0 ||
70b9f9 ........unhex(input[12]) < 0 || unhex(input[13]) < 0 ||
69b9b9 ..........unhex(input[15]) < 0 || unhex(input[16]) < 0 ||
7862f1 ...} EXIT_ERR("Bad input (addr, chipId, data must be hex)\n");
ffddf1c ...
d4a5f5a ...
95b00f ..} reset ....= input[0]= '0';
6a2502 ...} boardEn ....= input[1]= '0';
7c66a ...} ale ....= input[2]= '0';
b92d6a ...} adrsel1 ....= input[3]= '0';
8111b -*web.....= input[43] '0';
71d971 -*rdb.....= input[53] '0';
060751 -*adrlsel2 = input[63] '0';
3c1285 -*allactin = input[73] - '0';
449a0b -*addr .....= 16*unhex(input[9]) + unhex(input[10]);
3842c3 -*chipId .....= 16*unhex(input[12]) + unhex(input[13]);
2d9e2a -*data .....= 16*unhex(input[15]) + unhex(input[16]);
0f0ef6)
0c0af5a
0c0af5a
62d93f /* Decodes a hex char or returns -1 if bad. */
8d8462 static int unhex(char c) {
6b53c4 if ((c >= '0' && c <= '9'))
5f06d6 /* return (c - '0');
2f8d81 /* if (c == 'a' && c <= 'f')
3f06da /* return (c - 16 + 10);
1a49e3 /* if (c == 'A' && c <= 'F')
86a66f /* return (c - 'A' + 10);
8f12d4 /* return (-1);
956f5a
f4af5a
7baf5a
1baf5a
453e5f /*
dea850 /* Run the system for one clock cycle and update the state.
8945a5 /*
4442ee void RunClock(void) {
5c6725 /* int i, j, k, b;
5dd70b /* unsigned char key[7], m[8], c[8];
77af5a
9c9101 /* for (i = 0; i < 24; i++) {
84543c /* if (STARTUP_DELAY[i] > 0) {
604550 /* STARTUP_DELAY[i]--; 
3a4050 /* if (STARTUP_DELAY[i] == 0)
340ea6 /* STARTUP_DELAY[i] = -1; ........../* prevent stop if 1st C0=match */
866ef7 ........
76df1c ....
14af5a
586df6 /* DES CLOCK 5: Plaintext vector result from last DES is ready. */
1e94afe /* if (DES_POSITION == 5) {
8e23cf /* for (i = 0; i < 24; i++) {
0c6d1c /* k = 0; ........../* i = search engine ........../
5bd642 /* for (j = 0; j < 8; j++) {
0a018a /* b = WORKING_PTXT[8*i+j]; ........../* j = byte idx ........../
3c91f5 /* if ((STATE[0b8] & (1 << (b*8))) ......./* check plaintext vector ........../
5ce346c /* k = (k >> 1); 128; ........../* match = load 1 in k msb ........../
4e3059 /* else ...................../*
040eb7 /* k = (k >> 1); 0; ........../* no match = load 0 into k ........../
5a425c ........
8f3f33 /* k := STATE[REG_PTXTBYTE_MASK]; ........../* set bits where bytemask=1 ........./*
711a90 /* MATCH0 = (unsigned char)((k == 255) ? 1 : 0); 
0a15a5 ........
5dd663 /* if ((STATE[REG_SEARCH_STATUS(i)] & 1) == 0 || STARTUP_DELAY[i] > 0) {
0b983a /* if search not active, key delta = 0 and do C0 next */
e3472c /* WORKING_DELTA[i] = 0;
3dde9d /* WORKING_NEXT_SELECTOR[i] = 1;
3f73f0 /* else if (k != 0xff) ((STATE[REG_SEARCH_STATUS(i)] & 2) ||
4b09fe /* WORKING_DELTA[i] < 0) {
286722 /* If no match or CURRENTLY doing C1 or first DES result,
3d5a5c /* key delta = 1 and do C0 next;
85afaf /*
781df0 /* WORKING_DELTA[i] = 1;
8bb30b /* WORKING_NEXT_SELECTOR[i] = 0;
ca2d9b /*
0d670b /* if (k == 0x5fF)
0ee247 /* if (k == 0xff && STARTUP_DELTA[i] < 0)
9b33e9 /* printKeyInfo(stderr, "ALERT: Match C0; will backup for C1 ", i);
286e96 /* else if (WORKING_LAST_SELECTOR[i] == 0) {
2d5b77 /*
4a6e9f /* WORKING_NEXT_SELECTOR[i] = 1;
4d835f /* printKeyInfo(stderr, "ALERT: Match C0; will backup for C1 ", i);
/*\n * STARTUP-DELAY\n * \n */
 memcpy(NEXT~KEY\n WOR\n 8 =
 /*
8
*/
 \WOR\nWOR\nWOR\n(DES-POSITION
/*
(DES-POSITION
key);
{\n<
=\nof
7);
WORKI
-WORKING-PTXTL8*i+7-j
else
/*
8
*/
```
0af5a
if (DES_POSITION == 15) {
fa2135 for (i = 0; i < 24; i++) {
ccc53f memcpy(TTHIS_KEY+i7, NEXT_KEY+i7, 7);
effe92 memcpy(NEXT_KEY+i7, STATE+REG_SEARCH_KEY(i), 7);
17f74 for (i = 0; i < 24; i++) {
7af5a
}
```

`\n6b2135 for (i = 0; i < 24; i++) {
1542cc)
```

`\n8b9ef if (STATE[REG_SEARCHINFO] & 2) ( \n62580 for \n85e2b ccc if (DES_POSITION == 15) {
661de
```
\n8d0af5a
```

`\n81bfcd /* END OF DES CYCLE: Extract results */
77eae if (DES_POSITION == CHL Marks) PER DES-1) {
66b2135 for (i = 0; i < 24; i++) {
7af5a
```

`\n825a0 for \n85e9f ccc if (DES_POSITION == 15) {
661de
```

`\n8c6a9 if (STATE[REG_SEARCHSTATUS(i) & 2] == 0) ( \n66f0b for
86e0b ccc if (DES_POSITION == 15) {
661de
```

`\n949a9 if (STATE[REG_SEARCHSTATUS(i) & 2] == 0) ( \n66f0b for
86e0b ccc if (DES_POSITION == 15) {
661de
```

`\n99f9c if (STATE[REG_SEARCHINFO] & 1) ( \n66f0b for
86e0b ccc if (DES_POSITION == 15) {
661de
```

`\n9bf4d /* Update ciphertext selector (state & last) */
569701 working_LAN_SELCETOR[i] = (STATE0x47+8*i & 2) ? 1: 0;
5ad73b STATE[0x47+8*i] & 0xFD; /* select ciphertext 0 */
25abad if (WORKING_NEXT_SELCETOR[i] | 1 == 1)
8eb9a9 STATE[0x47+8*i] = 2; /* then select c1 */
206fe7
```

`\n1df1c /*
80dca8 /* LAST DES CLOCK: Load in the updated key */
473b35 if (DES_POSITION == 14) {
47
```c
-4c9a 0007dd476568004000f Page 6 of sim.c

e02135 ....... for (i = 0; i < 24; i++) {
    b45d9d ... if (WORKING_KDELTAC[i] == 1) { ................./* if key delta = 1 */
753981i = increment32((STATE+REG_SEARCH_KEY(i));
423c ...} else if (WORKING_KDELTAC[i] == -1) { ................./* if key delta = -1 */
0a40 ... decrement32((STATE+REG_SEARCH_KEY(i));
406f ... 
}
def1c ...

d1af5a
a964e ... /* DES CLOCK 0: Latch in new working keys and working ciphertexts */
5c6cb ... if (DES_POSITION == 0) {
7f06 ... for (i = 0; i < 24; i++) { ................./* i = search engine */
34d9d /* pick between ctxt 0 and ctxt 1? */
d38ae ... if ((STATE[REG_SEARCH_STATUS(i)] & 2) == 0 & & STARTUP_DELAY[i] == 0)
7b9 ... memcy((WORKING_CTXT+8*i, STATE+REG_CIPHERTEXT0, 8); ......../* copy c0 */
13d ... else
4ff ... mempcy(WORKING_CTXT+8*i, STATE+REG_CIPHERTEXT1, 8); ......../* copy c1 */
1a6fe ... 
eef1c ...

0f5a

c82d95 ... /* Update ChipAllActive, board all active */
57519a ... j = 1;
a763d6 ... for (i = 0; i < 24; i++) {
bd95a ... j & STATE[0x47+i*8];
45597 ... j = (j & 1) ? 1 : 0;
5e5f4 ... STATE[REG_SEARCHINFO] &= (255-4); ......../* set ChipAllActive ...........*/
e632 ... if ((STATE[REG_SEARCHINFO] & 16) == 0) /* if board all active enable = 0 */
95df ... merge((STATE[REG_SEARCHINFO], 8); ......../* set board all active */
3b9c ... ALLACTIVE_OUT = ALLACTIVE_IN;
051bd ... else
b3 ... ALLACTIVE_OUT = ALLACTIVE_IN & j;
b45e2 ... STATE[REG_SEARCHINFO] &= (255-8); ......../* set board all active */
5c79 ... STATE[REG_SEARCHINFO] |= (8*ALLACTIVE_OUT); ......../* set board all active */
71a ... 
b34cd ... /* Do any pending updates and update DES cycle position */
52cf5 ... if (PENDING_UPDATE_ADDR1 == 0) {
728c6 ... STATE[PENDING_UPDATE_ADDR1] = PENDING_UPDATE_DATA1;
0237 ... PENDING_UPDATE_ADDR1 = PENDING_UPDATE_ADDR2;
c854 ... PENDING_UPDATE_DATA1 = PENDING_UPDATE_DATA2;
93e49 ... PENDING_UPDATE_ADDR2 = PENDING_UPDATE_ADDR3;
6a199 ... PENDING_UPDATE_DATA2 = PENDING_UPDATE_DATA3;
9f07 ... PENDING_UPDATE_ADDR3 = -1;
6cc7f ... DES_POSITION = (DES_POSITION + 1) % CLOCKS_PER_DES;
e8eef ... 

c9af5a ...

a4f5a ...

34d39e static void desDecrypt(unsigned char m[8], unsigned char c[8],
77f6b ... key[56], message[64];
5717e ... int i;
16af5 ... 
2dd545 #ifdef DEBUG
6b6e ... printf("DES DECYPTE(k=":); for (i=0; i<7;i++) printf("%02X",k[i]);
75e4d ... printf("c="); for (i=0; i<8;i++) printf("%02X",c[i]);
7d745 ... #endif
28af5a ...

9f81b ... for (i = 0; i < 56; i++) {
6e93 ... key[55-i] = ((k[i]/8) << (i & 7)) & 128); i = 0;
8f5c3 ... for (i = 0; i < 64; i++) {
0d5ed ... message[63-i] = (c[i]/8) << ((i & 7) & 128); i = 0;
3c59d ... DecryptDES(key, message, message, 0);
be1f7 ... for (i = 0; i < 8; i++) {
4d52 ... mess[i] = 0;
c35c3 ... for (i = 0; i < 66; i++) {
9d751 ... if (message[63-i]) {
42fe29 ... mess[i/8] |= 128 >> (i%8);
45af5a ...

72d545 #ifdef DEBUG
ea47c0 ... printf("=");
3d17 ... for (i = 0; i < 8; i++) {
7cdc ... printf("%02X",m[i]);
```
--a790 001522c46c680b4000f Page 7 of sim.c

3f6f7 --printf("", clk=%ld\n",CLOCK_COUNTER);
48754 #endif
1e6f5a
876f6f )
57a5f1a
26a5f5a
e3a5f5a
2eaf5a

dba6b0 static void printKeyInfo(FILE *outDev, char *preamble, int searchUnit) {
eac359 --fprintf(outDev, preamble);
b97d84 --printf(outDev, "(K=%02X%02X%02X%02X%02X%02X, clk=%ld, searchUnit=%d)\n",
d06f47 -----------------STATE[0x40+8*searchUnit+6],STATE[0x40+8*searchUnit+5],
fa184 -----------------STATE[0x40+8*searchUnit+4],STATE[0x40+8*searchUnit+3],
b166f8 -----------------STATE[0x40+8*searchUnit+2],STATE[0x40+8*searchUnit+1],
ceed -----------------STATE[0x40+8*searchUnit+0], CLOCK_COUNTER, searchUnit);
b3a5f5a
607f32 --printf(preamble);
b76f22 --printf("(K=%02X%02X%02X%02X%02X%02X, clk=%ld, searchUnit=%d)\n",
e66f47 -----------------STATE[0x40+8*searchUnit+6],STATE[0x40+8*searchUnit+5],
f6a184 -----------------STATE[0x40+8*searchUnit+4],STATE[0x40+8*searchUnit+3],
ba166f8 -----------------STATE[0x40+8*searchUnit+2],STATE[0x40+8*searchUnit+1],
ceed -----------------STATE[0x40+8*searchUnit+0], CLOCK_COUNTER, searchUnit);
adaf5a
586f6f )
c0af5a
678f3f static void increment32(unsigned char *num) {
68708d --if (++(num[0]) == 0)
f2c3c0 --if (++(num[1]) == 0)
0949d0 --if (++(num[2]) == 0)
7754ed --++(num[3]);
e76f6f )
b8af5a
f3af5a
1fd6f2 static void decrement32(unsigned char *num) {
ded7c7 --if (num[0])-- == 0)
834f6b --if (num[1])-- == 0)
654ee --if (num[2])-- == 0)
251e5b --(num[3])--;
a0f6f )
44af5a
52a5f5a

#define REG_PTXT_VECTOR (0x00)
define REG_PTXT_XOR_MASK (0x20)
define REG_CIPHERTEXT0 (0x28)
define REG_CIPHERTEXT1 (0x30)
define REG_PTXT_BYTE_MASK (0x38)
define REG_SEARCHINFO (0x3F)
define REG_SEARCH_KEY(x) (0x40 + 8*(x))
define REG_SEARCH_STATUS(x) (0x47+8*(x))
define CLOCKS_PER_DES 16

int RunChip(char *input, FILE *outfile, int useRaw);
long getClockCounter(void);
int peekState(int reg); /* runs chip & returns DATA value */
--dفا5 000أ2c96768040011 Page 1 of testvec.c

8d23d3 /************************************************
8a1332a * testvec.c     **********************************************************
8b53d8f * DES ASCII Simulator, Test Vector Generation Program   **************
8a4959b * Written 1998 by Cryptography Research (http://www.cryptography.com)
5a58aaf * and Paul Kocher for the Electronic Frontier Foundation (EFF).
719b9f * Placed in the public domain by Cryptography Research and EFF.    *
5e499b * THIS IS UNSUPPORTED FREE SOFTWARE. USE AND DISTRIBUTE AT YOUR OWN RISK. *
9592eb * ---------------------------------------------------
15c755 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
5a29eb * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
65489b * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
6b29eb * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
86582 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
602fb5 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
31dec8 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
30b8d9 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
6329eb * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
9a59f8 * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
6e829eb * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
7f489b * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
63209b * IMPORTANT: U.S. LAW MAY REGULATE THE USE AND/OR EXPORT   *
5679ef * REVISION HISTORY:
2a9b9f * Version 1.0: Initial release by Cryptography Research to EFF.
9b10e5 * Version 1.0: Initial release by Cryptography Research to EFF.
6aaa63 * Version 1.0: Initial release by Cryptography Research to EFF.
4aadd3 *******************************************
Chapter 7: Chip Simulator Source Code

-8051 0009599d7c48004001 Page 2 of testvec.c

4775e /*
ac775e /*
0ff2c /* • THESE FUNCTIONS CREATE AND MANAGE THE TEST VECTORS.
5b775e /* GIVE2 CREATE OF FILE-TOCHIP);
9a775e /* (argc
argv[2],
FILE-TOCHIP);
81775e /* CREATE OF FILE-TOCHIP);
2445d /*
ba5a
6a5a
b179bf void main(int argc, char **argv) {
5a7ab · unsigned char startKey[73], plaintextVector[32];
73632 · unsigned char plaintextXorMask[83];
ad17 · unsigned char ciphertext0[83];
fc93c · unsigned char ciphertext1[83];
81b4b · unsigned char plaintextByteMask;
5bccc · int useCBC, extraXor, randomVector;
b9f74 · long totalClocks;
ac92d0 · char buffer[512];
1aaf5a 0796ed · if (argc != 3 && argc != 4) {
c0795d · fprintf(stderr, "Command line: TO_CHIP.OUT FROM_CHIP.OUT [RAW]\n");
5626d6 · fprintf(stderr, "TO_CHIP.OUT ... File for data going to chip\\n\n");
8134ad · fprintf(stderr, "... (if this file exists, it will be simulated.\\n\n");
141a4e · fprintf(stderr, "... otherwise, a new file will be created.\\n\n");
67803c · fprintf(stderr, "FROM_CHIP.OUT ... File for chip's output\\n\n");
703172 · fprintf(stderr, "RAW ... Gives unix CRLFs & no header.\\n\n");
8bb646c · exit(1);
37d41c ·
16af5a 026000 */
802ba4 · /* Open files and set CREATING_VECTOR to:
3369bb · ....@reading TOCHIP file,
ccf37 · ....1=create TOCHIP from user input,
7fbbb4 · ....2=generate random vector
def9a6 ...
18bf19 · OpenFiles(argc[1], argv[2], (argc == 4) ? 1 : 0);
43a5f5a 97ee0b · if (CREATING_VECTOR == 0) {
ac50ab · fprintf(stderr, "Using input vector from file.\\n\n");
e8eb83 · while (!) {
66b843 · if (fgets(buffer, 500, FILE_TOCHIP) == NULL)
52d5f5a · break;
01563f · if (strlen(buffer) < 10)
1a2d5f · break;
97f0ee · RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
b86f77 ...}
449d8 ... else {
899a2b · GetUserInfo(plaintextVector, plaintextXorMask, ciphertext0, ciphertext1, 
08828 ... &plaintextByteMask, &useCBC, &extraXor, &randomVector, startKey, 
f49e4 ... &totalClocks);
ac3a38 · if (randomVector == 0) {
1f35be · fprintf(stderr, "Seed=random (time-based)\\n\n");
b4d40 · srand((unsigned) time(NULL));
95c26c · HARDWIRED_CHIP_ID = (unsigned char)(rand() & 255);
717a5a ... else if (randomVector == 1) {
1a10a2 · fprintf(stderr, "Using user params.\\n\n");
e16a79 ... else {
5ac986 · fprintf(stderr, "Seed=%d\\n\n", randomVector);
cfccdd · srand(randomVector);
97c26c · HARDWIRED_CHIP_ID = (unsigned char)(rand() & 255);
45fe7 ...}
33af5a 21f70d ... /* Reset chip and set the chip ID */
9e9a5d · sprint(buffer, "01011111 00 002X 0000\\n\n", HARDWIRED_CHIP_ID);
5d5a5a · RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO); fputs(buffer, FILE_TOCHIP);
3d5a5a · RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO); fputs(buffer, FILE_TOCHIP);
cd5a5a · RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO); fputs(buffer, FILE_TOCHIP);
765a5a · RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO); fputs(buffer, FILE_TOCHIP);
0f73b · sprint(buffer, "11011111 002X 0000\\n\n", HARDWIRED_CHIP_ID, 
9f116b · HARDWIRED_CHIP_ID);
1eb67c · RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
95084e · fputs(buffer, FILE_TOCHIP);
--02cc 00029e33ba58040011 Page 3 of testvec.c

ba5b1f .......buffer[2] = '1';
34b07c .......RunChip(buffer, FILE_FROMCHIP, USE_RAW_10);
7a08e4 .......fputs(buffer, FILE_TOCHIP);
ce47a4 .......buffer[2] = '0';
ec3b67c .......RunChip(buffer, FILE_FROMCHIP, USE_RAW_10);
8008e4 .......fputs(buffer, FILE_TOCHIP);
b0a5f5 ...
fd317c .......if (randomVector == 1) {
971b90 .........LoadState(plaintextVector, plaintextXorMask, ciphertext0, ciphertext1,
4b91ef .............plaintextByteMask, useCBC, extraXor, startKey);
6d3cb4 .........proceedNormal(totalClocks);
916a79 .........else {
92057c .........proceedRandom();
7f6fe7 ..............
4bdf1c .......
}
a7a5f6 ......./* Clean up a bit (doesn't really matter -- this is test code :-) */
f9d9918 .......fclose(FILE_FROMCHIP);
a96d5b .......fclose(FILE_TOCHIP);
0def66 }

b7a5f5
ba4a5f5
b3b6db void proceedNormal(long totalClocks) {
3be2b8 ...long numClocks = getClockCounter();
341614 ...
69b29f .......int i,j,r;
8a05f5 ...
27a415 .......while (++numClocks < totalClocks) {
4a0a63 ......r = RunSimulator_CheckRegister(REG_SEARCDINFO);
021f6e .......if (r & 4) {
2e3b4b ......fpfmt(stderr, "--- Idle ---\n"");
318957 ...RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */
d7bdac ......fpfmt(stderr, "\\n",
3c06ab ......RunSimulator_DummyIO();
3256f6 ...
9b6fe7 ...........
442135 .......for (i = 0; i < 24; i++) {
9a4ae6 ......./* If we're going to see a stall, give some settling time */

59bb76 .......goodKey[6] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 0);
4c6426 .......goodKey[5] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 1);
c910b2 .......goodKey[4] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 2);
2cb297 .......goodKey[3] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 3);
9b14ef .......goodKey[2] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 4);
15b5ef .......goodKey[1] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 5);
1a2f2b .......goodKey[0] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 6);
fb4a5f ...
ab1f35 ...
ba5b1f ...
64677e .........for (j = 0; i < 7; j++) {
652f9e ......fpfmt(stderr, "%02X", goodKey[j]);
1def87 ......fpfmt("%02X", goodKey[j]);
9ef7b0 ...........
0dde35 ......fpfmt(stderr, "\n");
e10493 ......fpfmt("\n");
d7bf5ed ...RunSimulator_DummyIO(); ............../* Settling time */
6afe40 ...RunSimulator_DummyIO(); ............../* Settling time */
e336ec ...RunSimulator_SetRegister(REG_SEARCH_STATUS(1), 1); ............../* restart */
4242cc ...........
7f6fe7 ...........
06df1c ..................f1ef6e6
doef5a 
0ba5f5 
d31874 void proceedRandom(void) {
84a655 ...unsigned char readout[256];
74a4ed ...unsigned char goodKey[7];
29e15a ...int i;
```c
4afe41 - unsigned char plaintextVector[32];
14d1fd - char buffer[256];
23af5a - /* chip has already been set and the chip ID has been loaded */
73af5a
42c34a - /* Create plaintext vector with 181 bits set */
ae47e - memset(plaintextVector, 0, sizeof(plaintextVector));
70a328 - i = 0;
0d13b6 - while (i < 181) {
926486 - j = rand() & 255;
21e479 - if ((plaintextVector[j] / 8) & (1 << (j % 8))) == 0) {
54c749 - plaintextVector[j / 8] |= (1 << (j % 8));
52079a - i++;
2f6fe7 - ...
}
dcfd1c - ...
84af5a
68fd4c - /* Load state */
03393a - for (i = 0; i < 32; i++)
579479 - RunSimulator_SetRegister(REG_PTXT_VECTOR + 1, plaintextVector[i]);
c61f76 - for (i = 0; i < 8; i++)
d332be - RunSimulator_SetRegister(REG_PTXT_XOR_MASK + i, rand() & 255);
671f76 - for (i = 0; i < 8; i++)
cec0be - RunSimulator_SetRegister(REG_CIPHERTEXT0 + i, rand() & 255);
784e9e - RunSimulator_SetRegister(REG_CIPHERTEXT1 + i, rand() & 255);
e1c5ca - RunSimulator_SetRegister(REG_PTXTBYTE_MASK, 1 << (rand() & 7));
c99a9c - i = (rand() % 3) + (rand() & 16); /* 0/1/2 for CBC & extraXor. 16 = activ0n */
8b243d - sprintf(stderr, "Using mode %d with ActiveOn=%d.\n", (i183), i16);
02bf25 - RunSimulator_SetRegister(REG_SEARCHINFO, i);
040688 - for (i = 0; i < 24; i++) { /* set random start key */
891ca - for (j = 0; j < 7; j++)
5f1d1c - ...
}...daaf5a
8ea083 - /* Read out all registers (real and not) except for ptxt vector */
df190 - for (i = 0; i < 32; i++)
6b1901 - readout[i] = RunSimulator_CheckRegister(i);
059997 - /* Change the key in any stopped units */
219101 - for (i = 0; i < 24; i++)
35816f - if ((readout[REG_SEARCH_STATUS(i)] & 1) == 0) ............./* stalled? */
3d6ca2 - RunSimulator_SetRegister(REG_SEARCH_KEY(i),
65571a - readout[REG_SEARCH_KEY(i)] ^ 0x08);
51df1c - /* fix key */
151901 - readout[i] = RunSimulator_CheckRegister(i);
80b09e - /* scan stopped units */
53910f - for (i = 0; i < 24; i++) {
f12abb - if ((readout[REG_SEARCH_STATUS(i)] & 1) == 0) ............./* stalled? */
97e66a - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 0);
b1093a - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 1);
2179a3 - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 2);
d2df8b - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 3);
cf9d9f - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 4);
1536a3 - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 5);
3b4237 - goodKey[i] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i) + 6);
1d0776 - if (rand() % 8)
8436ec - RunSimulator_SetRegister(REG_SEARCH_STATUS(i), 1); ......../* restart */
53ca8d - fprintf(stderr, "***** full match in unit %d; extracted k = ", i);
02b0df - (j = 0; j < 7; j++) {
83ad3b - fprintf(stderr, "%02X", goodKey[j]);
b90c9 - printf("%02X", goodKey[j]);
8642cc - c05501 - fprintf(stderr, "\n");
de6fe7e - cdf1fc - ...
}...fbaaf5a
ab917c - /* pick a different chip, read/write some registers, and reset chip id */
3f4ef7 - do (i = rand() & 255; ) while (i == HARDWIRED_CHIP_ID);
308b20 - sprintf(buffer, "0x1011111 %02X %02X 00\n", i, HARDWIRED_CHIP_ID);
262f1f - RunChip(buffer, FILE_FROMCHIP, USE_RAW_I0);
```
--5543 0004f889734b04040011 Page 5 of testvec.c

477f59  ..puts(buffer, FILE_TOCHIP);
146c5f  ..buffer[2] = '1';
8921ff  ..RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
527f59  ..puts(buffer, FILE_TOCHIP);
077f59  ..buffer[2] = '0';
a21ff  ..RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
b07f59  ..puts(buffer, FILE_TOCHIP);
6b1ff6  ..for (i = 0; i < 8; i++)
feca76  ....RunSimulator_SetRegister(rand() & 255, rand() & 255);
161f76  ..for (i = 0; i < 8; i++)
466bcf  ....RunSimulator_CheckRegister(rand() & 255);
6615e8  ..sprintf(buffer, "%02X %02X 00\n", HARDWIRED_CHIP_ID,
04867c  ....HARDWIRED_CHIP_ID);
5b21ff  ..RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
a57f59  ..puts(buffer, FILE_TOCHIP);
836c5f  ..buffer[2] = '1';
ed21ff  ..RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
117f59  ..puts(buffer, FILE_TOCHIP);
c70e4  ..buffer[2] = '0';
7b21ff  ..RunChip(buffer, FILE_FROMCHIP, USE_RAW_IO);
1c7f59  ..puts(buffer, FILE_TOCHIP);
9daf5a
3074d9  /* Test board enable and ADRSEL1 */
234255  ..BOARD_EN_IN = 0;
71cb06  ..ADRSEL1_IN = 0;
732414  ..for (i = 0; i < 4; i++)
42ca76  ....RunSimulator_SetRegister(rand() & 255, rand() & 255);
0e2414  ..for (i = 0; i < 4; i++)
08fbcf  ....RunSimulator_CheckRegister(rand() & 255);
594255  ..BOARD_EN_IN = 0;
9891da  ..ADRSEL1_IN = 1;
241f76  ..for (i = 0; i < 8; i++)
71ca76  ....RunSimulator_SetRegister(rand() & 255, rand() & 255);
7b1f76  ..for (i = 0; i < 8; i++)
54fbcf  ....RunSimulator_CheckRegister(rand() & 255);
2c1889  ..BOARD_EN_IN = 1;
7fcb06  ..ADRSEL1_IN = 0;
da1f76  ..for (i = 0; i < 8; i++)
eaeca76  ....RunSimulator_SetRegister(rand() & 255, rand() & 255);
e41f76  ..for (i = 0; i < 8; i++)
b8bfce  ....RunSimulator_CheckRegister(rand() & 255);
f81889  ..BOARD_EN_IN = 1;
1491da  ..ADRSEL1_IN = 1;
4caf5a
8ce7e0  /* Make a final pass reading all the registers */
99741b  ..for (i = 255; i >= 0; i--)
771901  ..readout[i] = RunSimulator_CheckRegister(i);
7e889e  /* scan stopped units */
199101  ..for (i = 0; i <= 24; i++)
b02abb  if ((readout[REG_SEARCH_STATUS(i)] & 1) == 0) {
6e66a  ....goodKey[6] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+0);
c2093a  ....goodKey[5] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+1);
285d4a  ....goodKey[4] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+2);
96df8b  ....goodKey[3] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+3);
43df93  ....goodKey[2] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+4);
eb36a3  ....goodKey[1] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+5);
c34237  ....goodKey[0] = RunSimulator_CheckRegister(REG_SEARCH_KEY(i)+6);
9bc697  ..RunSimulator_SetRegister(REG_SEARCH_STATUS(i), 1); /* restart */
5aca8d  ..printf(stderr, "***** Full match in unit %d; extracted k = ", i);
34b8df  ..for (i = 0; j < 7; j++)
883d3b  ....fprintf(stderr, "%02X", goodKey[j]);
fc05c9  ....printf("%02X", goodKey[j]);
f7f22c  ....}
8d85501  ..printf(stderr, "\n");
a56fe7  ....}
04df1c  ..}
81e6f6  }
b2af5a
8baf5a
dbaf5a
be7897 void GetUserInfo(unsigned char plaintextVector[32],
4f2461  ..unsigned char plaintext XOR Mask[8];

Chapter 7: Chip Simulator Source Code
Chapter 7: Chip Simulator Source Code

---53eb 000af645a6880040011 Page 6 of testvec.c

261fa8 ...........unsigned char ciphertext[0];
24944e ...........unsigned char *plaintextByteMask, int *useCBC, int *extraXor,
8b9f02 ...........int *randomVector, unsigned char startKey[7], long *totalClocks) {
8e0e71 .......char buffer[1024];
98c6d6 .......unsigned char tmp[512];
fe17e0 .......int i;

b5a5fa
77659f
10766e
13b72a
4953dd
b0812c
4813bc
83f545
24f3bf
96a5fa
51ee8b
14766e
2db72a
aa8156
fba670
91b567
b9a5fa
b83a8c
b6766e
67b72a
a28156
c86f62
29f8f3
16a5fa
e9a5d9
eb766e
26b72a
541856
21cb49
1ef16a
def221
9f766e
b5b72a
96b448
69f706
ede2b5
7ca5fa
bd96d5
28766e
3ab72a
915b59
7a6c75
8e2e53
92a5fa
2e04d5
35766e
3eb72a
a6b448
6fb66
9db530
d4a5fa
40c8bf
a2766e
07e684
3c3a54
f30ed3
6ca5fa
aff4f6

printf("Enter plaintextVector values: ");
gets(buffer);
printf("Enter ciphertext 0: ");
gets(buffer);
printf("Enter ciphertext 1: ");
gets(buffer);
printf("Enter useCBC (0 or 1): ");
gets(buffer);
printf("Enter extraXor (0 or 1): ");
gets(buffer);
printf("Enter randomVector (0=randomize, 1=user input, >1=seed): ");
gets(buffer);
printf("Enter starting key: ");
gets(buffer);
printf("Enter number of clocks: ");
```c
74776e  ... gets(buffer);
775752  ... sscanf(buffer, "%ld", totalClocks);
0f9f80  ... if (*totalClocks < 1 || *totalClocks > 100000000L)
e4b093  ... EXIT_ERR("Must have between 1 and 1 billion clocks.
33a5a
6218ed  ... printf("ln. PtxtVector = ", plaintextVector, 32);
5aa738  ... printf(" PtxtXorMask = ", plaintextXorMask, 8);
44ff6c  ... printf(" Ciphertext 0 = ", ciphertext0, 8);
913b57  ... printf(" Ciphertext 1 = ", ciphertext1, 8);
83dd31  ... printf(" PtxtByteMask = ", plaintextByteMask, 1);
37e15b  ... printf(" ....useCBC = %d\n", useCBC);
012f30  ... printf(" ....extraXor = %d\n", extraXor);
be751  ... printf(" ....randomVector = %d\n", randomVector);
b81f55  ... printf(" ....starting key = ", startKey, 7);
bed787  ... printf(" ....Total clocks = %ld\n", totalClocks);
e2ef6e  )
e0a5f3
711884 void LoadState(unsigned char plaintextVector[32],
182461  ... unsigned char ciphertext0[8], unsigned char ciphertext1[8],
e3a024  ... plaintextByteMask, int useCBC, int extraXor,
40e619  ... unsigned char startKey[73]) {
c91e7e  ... int i;
0ba5f3
af539a  ... for (i = 0; i < 32; i++)
al9479  ... RunSimulator_SetRegister(REG_PTXT_VECTOR + i, plaintextVector[i]);
371f76  ... for (i = 0; i < 8; i++)
c0a554  ... RunSimulator_SetRegister(REG_PTXT_XOR_MASK + i, plaintextXorMask[7-i]);
131f76  ... for (i = 0; i < 8; i++)
6db0b6  ... RunSimulator_SetRegister(REG_CIPHERTEXT0 + i, ciphertext0[7-i]);
031f76  ... for (i = 0; i < 8; i++)
02abab  ... RunSimulator_SetRegister(REG_CIPHERTEXT1 + i, ciphertext1[7-i]);
05ea2e  ... RunSimulator_SetRegister(REG_PTXT_BYTE_MASK, plaintextByteMask);
5b35f1  ... RunSimulator_SetRegister(REG_SEARCHINFO, (useCBC ? 1 : 0) | 7e82a4  ... (extraXor ? 2 : 0) | 16);  ... enable board active */
9d0688  ... for (i = 0; i < 24; i++) {  ... for each engine */
c40441  ... RunSimulator_SetRegister(REG_SEARCH_KEY(i)+1, startKey[E3]);
da9b82  ... RunSimulator_SetRegister(REG_SEARCH_KEY(i)+2, startKey[E4]);
d432d7  ... RunSimulator_SetRegister(REG_SEARCH_KEY(i)+3, startKey[E5]);
751b04  ... RunSimulator_SetRegister(REG_SEARCH_KEY(i)+4, startKey[E6]);
a369d6  ... RunSimulator_SetRegister(REG_SEARCH_KEY(i)+5, startKey[E7]);
b8d4ae  ... RunSimulator_SetRegister(REG_SEARCH_KEY(i)+6, (startKey[0] + 1) & 255);
e7fd9  ... RunSimulator_SetRegister(REG_SEARCH_STATUS(i), 1);
8af1c  ... }  ... }
54ef6e  }
15af5a
5da5f3
ada5f3
bb5194 void RunSimulator_SetRegister(int addr, int data) (  
abd1fd  ... char buffer[256];
f8a5f3
3bd9b5  ... /* RESET_BOARD_EN,ALE,ADRSEL1,WRB,RDB,ADRSEL2,ALLACTIVE_IN,ADDR,CHIP_ID,DATA */
d25f05  ... sprintf(buffer, "1D%010d%02x %02x %02x\n", BOARD_EN_IN, ADRSEL1_IN,
afa8bd  ... ALLACTIVE_IN, addr, HARDWIRED_CHIP_ID, data);  
7721ff  ... RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
047f59  ... fputs(buffer, FILE_TOCHIP);
d1af5a
6ecb50  ... sprintf(buffer, "1D%010d%02x %02x %02x\n", BOARD_EN_IN, ADRSEL1_IN,
938ad8  ... ALLACTIVE_IN, addr, HARDWIRED_CHIP_ID, data);  
c621ff  ... RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);  
3d7f59  ... fputs(buffer, FILE_TOCHIP);
44af5a
bb5f05  ... sprintf(buffer, "1D%010d%02x %02x %02x\n", BOARD_EN_IN, ADRSEL1_IN,
c628ad  ... ALLACTIVE_IN, addr, HARDWIRED_CHIP_ID, data);  
8821ff  ... RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
b57f59  ... fputs(buffer, FILE_TOCHIP);
6af5a
073d7  ... if ((rand() & 3L) == 0)
1debd9  ... ALLACTIVE_IN = 1-ALLACTIVE_IN;
e9ef6e  )
```
Chapter 7: Chip Simulator Source Code

---

Chapter 7: Chip Simulator Source Code

---

e0af5a
1aaf5a
562371 void RunSimulator_DummyIO(void) {
5fd1fd ...char buffer[256];
dd5f5c ...int i,b,addr,chip;
f0a5f5a
bb5a8 ...if ((rand() & 3) > 0) {
6df7c5 ...addr = rand() & 255;
7cb22 ...chip = (rand() & 7) ? HARDWIRED_CHIP_ID : (rand() & 255);
b04f2c ...b = (rand() & 7) ? 1 : 0;
116e4c .../*RESET_BOARD_EN,ALE,ADRSSEL1,WRB,RDB,ADRSSEL2,ALLACT_IN,ADDR,CHIP_ID,DATA*/
58d814 ...sprintf(buffer, "1%0111%0d %02x %02x 00\n", b, ALLACTIVE_IN, addr, chip);
34b67c ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
e1084e ...fputs(buffer, FILE TOCHIP);
b76b7c ...RunChip(buffer, FILE FROMCHIP, USE_RAW IO);
4a084e ...fputs(buffer, FILE TOCHIP);
dea7f6 ...sprintf(buffer, "%d0111005d %02x %02x 00\n", b, ALLACTIVE_IN, addr, chip);
26b67c ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
7d8d48 ...fputs(buffer, FILE_TOCHIP);
e2b67fc ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
b208e ...fputs(buffer, FILE_TOCHIP);
00dca8 ...sprintf(buffer, "%d011110d %02x %02x 00\n", b, ALLACTIVE_IN, addr, chip);
96b76c ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
a3084e ...fputs(buffer, FILE_TOCHIP);
dbb76c ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
d1084e ...fputs(buffer, FILE_TOCHIP);
6d6b7c ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
53084e ...fputs(buffer, FILE_TOCHIP);
114d88 ...else {
1ad2a6 ...sprintf(buffer, "11011111% FF %02x FF\n", ALLACTIVE_IN, HARDWIRED CHIP_ID);
6cd137 ...for (i = rand() & 7; i > 0; i--) {
7df0ee ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
3ad7ef ...fputs(buffer, FILE_TOCHIP);
09eafe7 ...
98d0f1c ...
feef6e }
50a5f5a
55f5a5a
4ca5f5a
aba166 unsigned char RunSimulator_CheckRegister(int addr) {
9299d0 ...unsigned char rval;
d71f1d ...char buffer[256];
72af5a
229186 .../*RESET_BOARD_EN,ALE,ADRSSEL1,WRB,RDB,ADRSSEL2,ALLACT_IN,ADDR,CHIP_ID,DATA*/
fbac2d ...sprintf(buffer, "%d01111005d %02x %02x 00\n", BOARD_EN_IN, ADRSEL1_IN,
e027b8e ...ALLACTIVE_IN, addr, HARDWIRED CHIP_ID /*no data*/);
4121ff ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
317f59 ...fputs(buffer, FILE TOCHIP);
09a5f5a
0faaf92 ...sprintf(buffer, "%d01111005d %02x %02x 00\n", BOARD_EN_IN, ADRSEL1_IN,
35278e ...ALLACTIVE_IN, addr, HARDWIRED CHIP_ID /*no data*/);
cf49fa ...rval=(unsigned char)RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
dc7f59 ...fputs(buffer, FILE TOCHIP);
daef5a
a2234b ...sprintf(buffer, "%d011111%02x %02x 00\n", BOARD_EN_IN, ADRSEL1_IN,
ac278e ...ALLACTIVE_IN, addr, HARDWIRED CHIP_ID /*no data*/);
3321ff ...RunChip(buffer, FILE_FROMCHIP, USE_RAW IO);
b77f59 ...fputs(buffer, FILE TOCHIP);
cca5f5a
2076c9 ...return (rval);
cc0e6e }
36af5a
ada5f5a
4b28a6 int unhex(char c) {
a53c6 ...if (c >= '0' & c <= '9')
f203d6 ...return (c - '0');
618db1 ...if (c >= 'a' & c <= 'f')
180ada ...return (c - 'A' + 10);
e49eb3 ...if (c >= 'A' & c <= 'F')
d7aa66f ...return (c - 'A' + 10);
9912d4 ...return (-1);
b2ee6e }
```c
7-43

Chapter 7: Chip Simulator Source Code

```

```c
--3d19 000c2e4763180040011 Page 9 of testvec.c
e0a5f5a
1aaf5a
4c4579 int hex2bin(char *hex, unsigned char *bin) {
d2cc2f int i = 0;
a66a41 int j = 0;
66a5f5a
688e6f /* Trim string if comments present */
078b28 if (strchr(hex, '#') != NULL)
a052e6 /*strchr(hex, '\') != 0;
d8a3ed if (strchr(hex, '*') != NULL)
2566d1 /*strchr(hex, '\') != 0;
d8a65f if (strchr(hex, '\') != NULL)
762462 /*strchr(hex, '\') != 0;
6caf5a
9447de for (i = 0; i < (int)strlen(hex); i++) {
d28e69 if (hex[i] >= '0' && unhex(hex[i]) < 0)
d3d27e /*EXIT_ERR("Bad hex digit encountered.\n");
81df1c */
27af5a
1b47de for (i = 0; i < (int)strlen(hex); i++) {
3445f0 if (hex[i] < '0')
875f6a continue;
51643e if (hex[i] >= '0' && hex[i + 1] >= '0') {
577a24 bin[i++] = (unsigned char)(unhex(hex[i]) * 16 + unhex(hex[i + 1]));
0d59a3 continue;
245f6a continue;
ab0e67 */
a3e16f if (hex[i] >= '0') {
339eae bin[i++] = (unsigned char)(unhex(hex[i]));
3d6e7e */
5df01c */
ecb1d2 return ();
8cef6e };
c2af5a
f2af5a
847b71 void printHexString(char *tag, unsigned char *data, int len) {
3d17e0 for (i = 0; i < (int)len; i++)
dba5f5a
449650 printf("%s", tag);
fc0c12 for (i = 0; i < len; i++)
20ed57 printf("%02x", data[i]);
8aee8 printf("\n");
6eef6e };
8aaf5a
24af5a
188b39 void OpenFiles(char *toChipFilename, char *fromChipFilename, int useRaw) {
e27959 FILE.TOCHIP = fopen(toChipFilename, useRaw ? "rb" : "r");
c01896 if (FILE.TOCHIP != NULL) {
9df126 creating_vector = 0;
44a9d8 else ()
335eff FILE.TOCHIP = fopen(toChipFilename, useRaw ? "wb" : "w");
c5d318 if (FILE.TOCHIP == NULL) {
445716 fprintf(stderr, "Can't open \"s\" for toChip file\n", toChipFilename);
e4c12d exit(1);
d56fe7 */
0c48bc creating_vector = 1;
69df1c */
eba5f5a
7da670 FILE.FROMCHIP = fopen(fromChipFilename, useRaw ? "wb" : "w");
8d870f if (FILE.FROMCHIP == NULL) {
9b42e1 fprintf(stderr, "Can't open \"s\" for fromChip file\n", fromChipFilename);
25646c exit(1);
f0d1c */
7baf5a
35975d USE_RAW IO = useRaw;
7dc5de /* Activate this to add column descriptors in the output */
4e114a if (!useRaw) {
e382e3 fprintf(FILE.TOCHIP, "RESET\n");
4d7c15 fprintf(FILE.TOCHIP, "BOARD\n");
6f79b8 fprintf(FILE.TOCHIP, "::ALE\n");
da1111 fprintf(FILE.TOCHIP, "::ADRSEL\n");
c287b0 fprintf(FILE.TOCHIP, "::WRB\n");
```
--c2ba 001857ff29a80040011 Page 10 of testvec.c

93a32c ...fprintf(FILE_TOCHIP, "| | | | RDB\n")
71b3bd ...fprintf(FILE_TOCHIP, "| | | | ADRSEL2\n");
d017df ...fprintf(FILE_TOCHIP, "| | | | | | | ALLACTIVE_IN\n");
6bed08 ...fprintf(FILE_TOCHIP, "| | | | | | | ADDR\n");
173c97 ...fprintf(FILE_TOCHIP, "| | | | | | | \\ CHIP_ID\n");
f1102a ...fprintf(FILE_TOCHIP, "| | | | | | | \\ DATA\n");
4aea2f ...fprintf(FILE_TOCHIP, "| | | \// ALLACTIVE_OUT\n");
e1e4e9 ...fprintf(FILE_FROMCHIP, "DATA\n");
541499 ...fprintf(FILE_FROMCHIP, "\// ALLACTIVE_OUT\n");
4dc3ec ...fprintf(FILE_FROMCHIP, "\// -- IsActive [0..23] --\n");
a6df1c ...
187454 #endif
217c65 ...fprintf(FILE_FROMCHIP, "KEY ............ DES_OUT ............ MATCH & SELECT1:\n");
1ae6e6 }
85af5a
This chapter contains schematic diagrams of the printed-circuit boards that we designed and built for the DES Cracker. It also includes a few other details about the hardware.

Each hardware board holds 64 DES Cracker chips. In this schematic, we only show how 8 of the chips are wired. The rest are wired almost identically. Each “All Active Out” pin is daisy-chained to the next “All Active In” pin. The “Chip ID” pins on each chip are connected directly to either ground or power, to tell the chip its binary chip number among all the chips on the board. If you examine these pins for the eight chips shown, you’ll see how they change.

The boards fit into card-cages which are connected to each other and to the host computer by a 50-pin ribbon cable. The card-cages are modified Sun-4/470 server card cages. The modifications we made to their backplanes are detailed toward the end of the chapter.

**Board Schematics**

The schematics begin on the next page.
Sun-4/470 backplane modifications

The first DES Cracker uses several chassis recycled from Sun-4/470 servers to hold its boards. Each chassis contains a card cage, power supplies, fans, and covers. In the card cage there is a backplane, which is a printed circuit board that holds the connectors for each board that can be plugged into the card cage. Each row has connectors for 12 slots numbered from 1 to 12. The card cage is sized for “9U” VMEbus boards, each of which has three large 96-pin connectors. Therefore, the backplane also has three 96-pin connectors per board, called P1, P2, and P3. Each of these 96-pin connectors has three rows of 32 pins inside it, called Rows A, B, and C.

We modified the backplane as follows:

Top Row (P1): No modification. We just use this as a board holder. There is no signal from our boards to these connectors.

Middle Row (P2): No modification. We just use this as a board holder. There is no signal from our boards to these connectors.

Bottom Row (P3): Power and signaling for the DES Cracker boards, as follows:

Table 8-1: Signal assignments on bottom connectors

<table>
<thead>
<tr>
<th>Row A</th>
<th>Original Assignment</th>
<th>New Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin 1 to 25</td>
<td>+5 Volts</td>
<td>Supply voltage for DES Cracker chips</td>
</tr>
<tr>
<td>Pin 26 to 27</td>
<td>+12 Volts</td>
<td>Not used</td>
</tr>
<tr>
<td>Pins 28 to 29</td>
<td>-12 Volts</td>
<td>Not used</td>
</tr>
<tr>
<td>Pins 30 to 32</td>
<td>-5 Volts</td>
<td>Not used</td>
</tr>
<tr>
<td>Row B</td>
<td>Original Assignment</td>
<td>New Assignment</td>
</tr>
<tr>
<td>Pin 1</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
<tr>
<td>Pin 2</td>
<td>Reserved</td>
<td>Not used</td>
</tr>
<tr>
<td>Pin 3</td>
<td>Reserved</td>
<td>Reset (C_RST)</td>
</tr>
<tr>
<td>Pin 4</td>
<td>Reserved</td>
<td>Read Strobe (C_RDB)</td>
</tr>
<tr>
<td>Pin 5</td>
<td>Reserved</td>
<td>Write Strobe (C_WRB)</td>
</tr>
<tr>
<td>Pin 6</td>
<td>Reserved</td>
<td>Address Latch Enable (C_AEN)</td>
</tr>
<tr>
<td>Pin 7</td>
<td>Reserved</td>
<td>Control_1 (C_CNT1) or C_ADRSELB</td>
</tr>
<tr>
<td>Pin 8</td>
<td>Reserved</td>
<td>Control_2 (C_CNT2) or C_CS</td>
</tr>
<tr>
<td>Pin 9</td>
<td>Reserved</td>
<td>Data 7 (C_D7)</td>
</tr>
<tr>
<td>Pin 10</td>
<td>Reserved</td>
<td>Data 6 (C_D6)</td>
</tr>
<tr>
<td>Pin 11</td>
<td>Reserved</td>
<td>Data 5 (C_D5)</td>
</tr>
<tr>
<td>Pin 12</td>
<td>Reserved</td>
<td>Data 4 (C_D4)</td>
</tr>
<tr>
<td>Pin 13</td>
<td>Reserved</td>
<td>Data 3 (C_D3)</td>
</tr>
<tr>
<td>Pin 14</td>
<td>Reserved</td>
<td>Data 2 (C_D2)</td>
</tr>
<tr>
<td>Pin 15</td>
<td>Reserved</td>
<td>Data 1 (C_D1)</td>
</tr>
</tbody>
</table>
Table 8-1: Signal assignments on bottom connectors (continued)

<table>
<thead>
<tr>
<th>Pin</th>
<th>Assignment</th>
<th>New Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Reserved</td>
<td>Data 0 (C_D0)</td>
</tr>
<tr>
<td>17</td>
<td>Reserved</td>
<td>Address 7 (C_A7)</td>
</tr>
<tr>
<td>18</td>
<td>Reserved</td>
<td>Address 6 (C_A6)</td>
</tr>
<tr>
<td>19</td>
<td>Reserved</td>
<td>Address 5 (C_A5)</td>
</tr>
<tr>
<td>20</td>
<td>Reserved</td>
<td>Address 4 (C_A4)</td>
</tr>
<tr>
<td>21</td>
<td>Reserved</td>
<td>Address 3 (C_A3)</td>
</tr>
<tr>
<td>22</td>
<td>Reserved</td>
<td>Address 2 (C_A2)</td>
</tr>
<tr>
<td>23</td>
<td>Reserved</td>
<td>Address 1 (C_A1)</td>
</tr>
<tr>
<td>24</td>
<td>Reserved</td>
<td>Address 0 (C_A0)</td>
</tr>
<tr>
<td>25</td>
<td>Reserved</td>
<td>GND</td>
</tr>
<tr>
<td>26</td>
<td>Reserved</td>
<td>GND</td>
</tr>
<tr>
<td>27</td>
<td>Reserved</td>
<td>GND</td>
</tr>
<tr>
<td>28</td>
<td>Reserved</td>
<td>GND</td>
</tr>
<tr>
<td>29</td>
<td>Reserved</td>
<td>GND</td>
</tr>
<tr>
<td>30</td>
<td>Reserved</td>
<td>GND</td>
</tr>
<tr>
<td>31</td>
<td>Reserved</td>
<td>+5 V supply to all Interface ICs</td>
</tr>
<tr>
<td>32</td>
<td>Reserved</td>
<td>+5 V supply to all Interface ICs</td>
</tr>
</tbody>
</table>

Row C

<table>
<thead>
<tr>
<th>Original Assignment</th>
<th>New Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pins 1 to 25</td>
<td>GND</td>
</tr>
<tr>
<td>Pins 26 to 27</td>
<td>+12 Volts</td>
</tr>
<tr>
<td>Pins 28 to 29</td>
<td>-12 Volts</td>
</tr>
<tr>
<td>Pins 30 to 32</td>
<td>-5 Volts</td>
</tr>
</tbody>
</table>

Row A, pins 1-25 provide the supply voltage for the DES Cracker chips. The supply is normally +5 Volts.

The chips can be run on a lower voltage, to reduce power consumption and heat generation. In that case, two voltages must be supplied. The lower voltage for the DES Cracker chips is supplied on Row A, pins 1-25. +5 volts is supplied to the interface circuitry on Row B, pins 31 and 32. In low voltage operation, Jumper JP1 on each of the DES boards must be removed. If the DES chips are using +5 Volts, then no external power connects to Row B, pins 31 and 32, and Jumper JP1 on each of the DES boards is connected.

**Physical Modifications on P3 Bus (Bottom Row)**

The P3 bus (bottom row) of the backplane has 12 slots. Some of these slots are wired to their neighboring slots, forming a bus. In its original Sun configuration, the P3 bus was mainly used for a high-speed memory bus between the CPU board and the memory boards. It was divided into 4 independent groups:

**Group 1**

This group has 7 slots (from 1 to 7) which have their Row B’s bussed together.
Group 2
   This has only slot 8. Its Row B did not connect to any other.

Group 3
   This has only slot 9. Its Row B did not connect to any other.

Group 4
   This group has 3 slots (from 10 to 12) which have their Row B’s bussed together.

We modified the backplane to connect each of these four groups together, so that P3 Row B connects from slot to slot along the whole backplane.

On both slot 1 and slot 12 we added a dual-row header to the P3 connector, Rows B and C (signals and grounds), so that a 50-pin ribbon cable can connect to the bus. These headers allow each chassis to be cabled to the next chassis, and also allow the first chassis to be cabled to a general purpose computer, where the software that controls the DES Cracker runs.

On slot 11, we also added a dual-row header to the P3 connector, Rows A and B (Supply voltage and signals), to let us install termination resistors when no ribbon cable is attached to Slot 12. These protect the integrity of the signals on the bus.

**PC Interfaces**

The first chassis connects to the controlling computer via a ribbon cable, which attaches to the dual-row header installed on Slot 1. This cable leads to a plug-in hardware card which provides three parallel I/O ports. The software talks to this card, causing it to write commands to the ribbon cable, or read results back from the ribbon cable. The software runs in an ordinary IBM PC, and could be ported to other general purpose computers.

Our project used either of two interface cards. Both are from National Instruments Corporation of Austin, Texas, reachable at http://www.natinst.com or +1 512 794 0100. Their PC-AT bus interface card is called the PC-DIO-24, order number 777368-01. For laptops, a “PC card” (PCMCIA) interface is also available, the DAQCard-DIO-24, order number 776912-01. This card requires the PSH27-50F-D1 cable, with order number 776989-01.

Other parallel interface cards that provide 24 bit I/O could also be made to work.
Errata

This page contains notes about errors detected late in the hardware or software published herein.

Chip select for reading

The DES Cracker chips do not properly tristate their data buffers. When any chip on any board is reading, every other DES Cracker chip drives garbage onto its data pins. The buffer enables were not qualified by the Board Enable and Chip Enable signals. The initial hardware boards were modified to circumvent this by providing individual RDB signals to each chip, qualifying them externally with an FPGA. The correct fix is in top.vhd in the chip VHDL; near the last line, change:

```
DATA <= DATAO when (RDB = '0' and ADDSEL2 = '0') else (others => 'Z');
```

to:

```
DATA <= DATAO when (RDB = '0' and ADDSEL2 = '0' and CHIP_EN = '1')
                 else (others => 'Z');
```

This also involves adding CHIP_EN as an output of upi.vhd.
In This chapter:
• Abstract
• Introduction
• The basic idea
• Details of such a machine
• Obtained results and remarks
• Conclusion
• Acknowledgement

9

Breaking One Million DES Keys
by Yvo Desmedt

This paper was presented at Eurocrypt 1987 by Yvo Desmedt and Jean-Jacques Quisquater, under the title “An Exhaustive Key Search Machine Breaking One Million DES Keys”. We publish it here for the first time, since no proceedings were made. It points out some research directions in parallel brute force codebreaking that are still useful today.

Abstract

The DES is in the commercial and industrial world the most used cryptoalgorithm. A realistic exhaustive key search machine will be proposed which breaks thousands of keys each hour, when DES is used in its standard 8 byte modes to protect privacy. Also authenticity protection with DES is sometimes insecure.

Introduction

The DES is the NBS* and ANSI† standard for encryption. It has been proposed to become an ISO‡ standard, under the name DEA1. From the beginning Diffie and Hellman mentioned that one DES key could be broken under a known plaintext attack using an exhaustive keysearch machine.§ However the design was criticized because practical problems as size and power dissipation were not taken into

‡ “Data Encipherment, Specification of Algorithm DEA1”, ISO/DP 8227 (Draft Proposal), 1983
consideration. Hoornaert* proposed last year a realistic exhaustive keysearch machine, which solved all practical problems. Instead of breaking DES in half a day (as in the Diffie-Hellman machine), the cheap version ($1 million) needs maximum 4 weeks to find the key. In practice however companies or secret agencies want to break several keys at once. Indeed for doing industrial espionage, companies want to break as many communications as possible of their main competitors. Secret agencies want to be able to eavesdrop all communications and to follow up industrial developments in other countries which may be used for military purposes. The above machine is unpractical or expensive for this purpose. Instead of using thousands of machines for breaking thousands of keys, one modified machine is enough.

The basic idea

At first sight if one wants to break one million keys with an exhaustive machine one needs one million pairs (plaintext,ciphertext)=(Mi,Ci) and do the job for each different pair. If all these pairs have the same plaintext M, the exhaustive machine can do the same job by breaking all these one million ciphertexts, as in the case it had only to break one. This assumption is very realistic, indeed in letters some pattern as e.g. "Yours Sincerely" are common. For all standard† 8 bytes modes a partially known plaintext attack is sufficient. In the case of ECB a ciphertext only attack is sufficient. Indeed the most frequent combination of 8 bytes can easily be detected and used. Evidently more machines can handle more different plaintext patterns. So, a few machines can break millions of keys. The number of different patterns can be reduced by using a chosen plaintext attack!

Details of such a machine

Although we did not built it, in this section sufficient details are given to show that such a machine is feasible. The machine will be based on a small extension of the DES chips used in Hoornaert's machine. We will call the ciphertexts for which one wants to break the key: "desired" ciphertexts. In one machine, each of the (e.g.) 25 thousand DES chips will calculate ciphertexts for variable keys starting from the same 8 byte "plaintext" pattern. The machine has to verify if such a ciphertext is the same as some "desired" ciphertext. If so, it has to communicate the corresponding key to the Key Handling Machine (KHM) and the "number" of the "desired" ciphertext. However each used DES chip generates each second about

† "DES modes of operation", FIPS (NBS Federal Information Processing Standards Publ.), no. 81, Washington D.C., December 2, 1980
Details of such a machine

one million pairs (ciphertext, key). This gives a major communication problem. Indeed all this information (about 110Mbit/sec. = (56 key bits + 64 ciphertext bits) x 1M DES/sec.) cannot be communicated constantly outside the chip. To avoid this communication problem, the chip will internally exclude ciphertexts which certainly are not equal to a “desired” ciphertext. So only a fraction has to be communicated to the outside world. Hereto the “desired” ciphertexts were previously ordered based on their first 20 bits, which are used as address of the desired ciphertexts. If more than one of these “desired” ciphertexts have the same 20 first bits then one of them will later be transferred to the exhaustive machine. The others will be put on a waiting list. In the exhaustive machine bits of the desired ciphertexts are spread in RAMs, as explained later, using the 20 first bits as address. Each extended DES chip is put on a hybrid circuit together with 4 RAMs of 1Mbit and a refresh controller (see also fig. 1). For each enumerated key the DES chip communicates the 20 first bits of the corresponding generated ciphertext to the RAMs as address. The 4 bits information stored in the RAMs correspond to the next 4 bits of the desired ciphertexts. The RAMs communicate to the modified DES chip these 4 bits. Only if these 4 bits are equal to the corresponding ones in the generated ciphertext, the generated pair (ciphertext, key) is communicated outside the DES chip to a local bus (see fig. 1). So in average the communication rate is reduced, by excluding the ciphertexts which are certainly not desired. About 10 of these hybrids are put on a small PCB. A custom designed chip checks the next 10 bits (the bits 25 till 34) of the ciphertexts using the same idea as for the 4 bits (the bits 21 till 24). Hereto 10 RAMs each of 1Mbit are used, the address is again the first 20 bits of the generated ciphertext. Only if the check succeeds the pair (ciphertext, key) is communicated to the outside world via a global bus. This reduces the communication between the local bus and the global bus with a factor 1000. About 2500 similar PCBs are put in the machine. The last 30 bits of the ciphertext are checked further on. Hereto similar hardware controls several PCBs. Finally a small machine can do the final check. The machine KH M checks the correctness of the key on other (plaintext, ciphertext) pairs or on the redundancy in the language. Once each (e.g.) hour the machine KH M will update the broken keys and put the ones which are on the waiting list into the exhaustive machine (if possible). Suppose that one hybrid cost $80, then the price of $3 million (25,000 x hybrid + custom chips + PCBs + etc) for this machine is realistic.
Obtained results and remarks

The described machine breaks about one million keys in 4 weeks, or in average about 3000 keys each hour. By updating the broken keys better results can be obtained.* Practical problems as buffering, synchronization, MTBF, power dissipation, size, reloading of the RAMs and so on are solved by the author. Optimizations under several circumstances and variants of the machine are possible. In view of the existing rumors that a trapdoor was built in DES by NSA, the feasibility of this machine shows that a trapdoor was not needed in order to break it. Old RAM technology allowed to design similar (or larger) machines which break less keys (e.g. thirtytwo thousand keys). This attack can be avoided if the users of DES use the CFB one byte mode appropriately, or use new modes,† or triple encryption with two different keys. DES-like algorithms can be designed which are more secure against the described attack and which use a key of only 48 bit, and which have the same encryption/decryption speed as DES (if used with fixed key).‡ The protection of the authenticity of (e.g. short) messages with DES is sometimes insecure.§ These results combined with the above one, shows that the authentication of standardized messages with DES may be worthless. Remark finally that the DES chip used in this machine does not use the state of the art of VLSI. Indeed about only 10,000 transistors are used in it. Megabits RAMs are easily available.

Conclusion

Every important company or secret agency over the world can easily build such a machine. Because it is not excluded that such machines are already in use by these organizations, the author advises the users to be careful using DES. Because the most used modes are breakable, the users have to modify their hard- or software in a mode which avoids this attack. Meanwhile only low-sensitive information can be transmitted with DES. If the authenticity of the messages is protected with DES under its standardized use, short messages have to be enlarged.

† Quisquater, J.-J., Philips Research Laboratory, Brussels, paper in preparation.
Acknowledgement

The author is sponsored by the Belgian NFWO. The author is very grateful to F. Hoornaert, IMEC-ESAT, Leuven, and J.-J. Quisquater, Philips Research Laboratory, Brussels, for many suggestions and improvements.

Y.Desmedt
ESAT Laboratory
Katholieke Universiteit Leuven
Kard. Mercierlaan 94
B-3030 Heverlee, Belgium
This paper was written in Spring 1996. Its performance numbers are several years out of date, and it used what hardware was handy, rather than the best possible hardware for its time. Still, results based on actually building working devices are preferable to much better theories about reality.

Abstract

We examine issues in high-performance cryptanalysis, focusing on the use of programmable logic. Several standard techniques from computer architecture are adapted and applied to this application. We present performance measurements for RC4, A5, DES, and CDMF; these measurements were taken from actual implementations. We conclude by estimating the resources needed to break these encryption algorithms.

Introduction

Large-scale open electronic communications networks are spreading: for example, mobile computing is on the rise, the Internet is experiencing exponential growth, and electronic commerce is a hot topic. With these advances comes a need for robust security mechanisms, and they in turn depend critically on cryptographic protection. At the same time, computer power has been growing at dizzying rates,
matching or exceeding Moore’s Law. Therefore, in this rapidly changing environment, it is important to assess the strength of deployed encryption algorithms against the tremendous computational power available to potential adversaries.

The best attacks on today’s symmetric-key encryption algorithms simply apply massive computing resources to break their security by pure brute force. If a cryptographic algorithm is secure, it will be far too expensive for an attacker to gather the processing power necessary for such a brute-force cryptanalytic attack to succeed. Assessing the security of a cryptographic algorithm against this threat, then, involves surveying the state of the art in cryptanalytic computational power and estimating the investment required to mount this type of attack.

This paper explores the use of programmable logic hardware devices in cryptanalytic applications. Programmable logic attempts to provide much of the premier performance available from custom hardware, while partially retaining the reconfigurability and ease of development benefits found in software.

Our research draws heavily on the computer architecture field. Surprisingly, many techniques, tools, and models for the design of general-purpose processors also proved useful in the specialized domain of cryptanalytic hardware. We investigate the benefits of various forms of parallelism, including pipelining and superscalar architectures. We also examine and identify critical structural hazards and data hazards, as well as the crucial performance bottlenecks. This paper focuses especially on an analogue of the central “CPU time” formula from [20]. By framing the problem from the perspective of system architects, we were able to take advantage of the extensive knowledge base available in the architecture literature.

This paper is organized as follows. The section “Motivation” elaborates on the need for estimates of the performance of cryptanalytic hardware, and the section “Related Work” lists previous work which touches on this project and influenced our approach. Next, the the section “Technical Approach” introduces our experimental methodology and goals. The section “Design and Analysis” describes our design, implementation, and data in depth, providing a detailed technical analysis. Finally, the section “Future” briefly identifies some areas for future research, and the “Conclusion” concludes the paper.

**Motivation**

There is currently a strong need for a solid assessment of the resources required to break the common cryptographic algorithms. This information is a crucial data point for system designers—they need this information to determine which encryption algorithm is appropriate for their system. The need is only intensifying: weak encryption is becoming the norm, earlier assessments are either incomplete
or out-of-date, and steady increases in computing power are threatening the viability of these weak encryption systems.

Security is little more than economics. A cryptographic system is secure when it costs more to break it than the data it is protecting is worth. Accordingly, determining the strength of an encryption algorithm comes down to measuring the cost of the cryptanalytic resources needed to break the system. That explains the basic need for an evaluation of the cryptanalytic performance possible today.

In fact, several recent factors make the need more urgent. Weak encryption is being widely deployed. SSL with 40-bit RC4 is becoming a de facto standard for secure Web channels, largely because of Netscape’s support. GSM, a European mobile telephony system, depends for its link-layer security on A5, an apparently weakened algorithm. Export restrictions are largely to blame for the recent preponderance of weak encryption algorithms; they are an unfortunate fact of life at the moment. This intensifies the need for accurate estimates of the true protection these cryptographic algorithms offer. For extremely strong algorithms, it is sufficient to provide order-of-magnitude estimates to show that breaking these algorithms requires absurd collections of resources; but when it is feasible (or barely feasible) to break an encryption algorithm, it becomes extremely important to pinpoint the cost of cryptanalysis accurately.

The section entitled “Related Work” lists several earlier algorithm assessments. DES has received by far the most attention, but we are also greatly interested in the (today all-too-common) case of exportable encryption algorithms. Most of the experience with weak encryption systems has been with software cryptanalysis; yet programmable logic may be the most cost-effective method of assembling computational power for this problem. A recent paper [4] did briefly address the cost-effectiveness of programmable logic, but their estimate appears to be based on flawed assumptions. The one work which investigated the problem most closely [22] was a good start, but it didn’t go far enough: their estimates were based on theoretical calculations, instead of real implementations and measurements.

Therefore, there is new ground to cover, and previous work to validate. We will explore the applicability and performance of programmable logic to cryptanalysis of A5, DES, CDMF, and RC4. This paper attempts to provide a solid, rigorous assessment of the economics of cryptanalysis, relying on actual implementations and experimental measurements.
**Related work**

Previous exploration into exhaustive keysearch has tended to concentrate on either software implementations or custom hardware designs; not much has been reported on FPGA (programmable logic) architectures. We will survey the results available in the open literature.

The first public brute-force cryptanalysis of 40-bit exportable RC4 appeared from the Internet cypherpunks community. (The NSA (National Security Agency) had almost certainly mounted an exhaustive 40-bit search of RC4 long before that, but they're playing their cards close to their chest.) The cypherpunks are a loose-knit community dedicated to exploring the social ramifications of cryptography. To demonstrate the need for more secure encryption, Hal Finney challenged his fellow cypherpunks to break 40-bit RC4 [16]. Soon Adam Back, David Byers, and Eric Young announced [3] that they had successfully searched the 40-bit keyspace with a software implementation running on the idle cycles of several workstations. At the same time, Damien Doligez had also independently finished a successful sweep of the RC4 40-bit keys [12], with the same software implementation. Not long later, Piete Brooks, Adam Back, Andrew Roos, and Andy Brown organized a distributed effort [5] which used donated idle cycles from many machines across the Internet to finish a second challenge in 31 hours, again using a similar software implementation. The cypherpunks efforts gave us a fairly accurate estimate of the complexity of exhaustively searching the RC4 40-bit keyspace in software.

There have been no reports of any experience with exhaustive keysearch of A5 in the open literature. The details of the A5 algorithm were only recently revealed to the public [1], so it is perhaps not surprising that it has received less attention. Several cryptographers' initial reaction was that there must be a trivial brute-force attack on A5 requiring $2^{40}$ operations [26],[1]. No such attack ever materialized, and it became clear that the matter was not so trivial as initially imagined [26],[2]. The current consensus appears to be that A5's strength is possibly somewhat more than a 40-bit cipher but less than its 64-bit key might indicate.

There have not been any reports on CDMF exhaustive keysearch in the literature, either. On the other hand, CDMF is very similar to DES—it is essentially DES with a reduced 40-bit keylength—so all the research into understanding DES keysearch will apply immediately to CDMF. As we shall see, there has been extensive work examining DES brute-force cryptanalysis.

There have been many studies into the economics of a DES keysearch implementation in custom hardware. (No one has seriously proposed breaking DES via software, as general-purpose computers are orders of magnitude slower at this task than specialized hardware.) The earliest estimate came not long after DES was ratified as a national standard. Whit Diffie and Martin Hellman designed a system...
containing a large number of custom-designed chips [11]. They estimated that their $20 million architecture could recover a DES key each day. After their paper appeared, great controversy ensued. Some argued that the mean time between failures would be inherently so small that the machine could never work; Diffie and Hellman refuted these objections, although they also increased their cost estimate somewhat [27], p.283. After the controversy died down, the final estimate was that DES would be insecure by the year 1990 [19]. A later paper suggested that a $1 million custom-designed hardware architecture could break DES in 9 days with technology forecasted to be available by 1995 [18]. Another more recent estimate took advantage of an extremely fast DES chip (designed for normal cryptographic use, not cryptanalysis), concluding that a $1 million assembly could search the DES key space in 8 days [31],[13],[14]. Yet another study examined the feasibility of using existing general-purpose content-addressable processors, and concluded that a DES keysearch would take 30 days on them with a $1 million investment [30] Even more writing on the subject of hardware DES keysearch can be found in [25], and some issues in DES chip design can be found in [21],[15],[6].

All these estimates were superseded by a compelling 1993 paper [31] from Michael Wiener. He went to the effort of assembling a very comprehensive design (extending for a hefty 42 pages!) of a custom-hardware DES keysearch machine, including low-level chip schematics as well as detailed plans for controllers and shelving. After a $0.5 million investment to design the machine and $1 million to build it, a DES key could be recovered each 3.5 hours, he argued. (Note the large development cost. This is a unique attribute of custom hardware designs.) His work has remained the definitive estimate of DES keysearch cost since then. On the other hand, we have seen 3 years of steady progress in chip performance and cost since then, and Moore's law remains as true as ever, so Wiener's figures should be adjusted downward accordingly.

This year an ad-hoc group of experts was convened to recommend appropriate cryptographic key lengths for corporate security; their report [4] was very influential. In this larger context, they very briefly surveyed the application of software, reconfigurable logic, and custom hardware to the brute-force cryptanalysis of 40-bit RC4 and (56-bit) DES. We are a bit skeptical about the precise performance predicted for an RC4-cracking chip: they claimed that a single $400 FPGA ought to be able to recover a 40-bit RC4 key in five hours. (Amortizing this over many keysearches, they determined that each keysearch would cost $0.08, causing some to refer to 40-bit RC4 as "8-cent encryption".) This estimate seems extremely optimistic, as it would require 30 million key trials per second; RC4 key setup requires at least 1024 serialized operations (256 iterations of a loop, with 4 memory accesses and calculations per iteration), so this would represent a throughput of 30 billion operations per second. Even with a dozen parallel independent keysearch
engines operating on the chip (which would require serious hardware resources), this would imply clock rates measured in Gigahertz—a rather unlikely scenario! Accordingly, our skepticism helped motivate us to attempt an independent investigation of these issues.

At the other extreme, we are also concerned about gross overestimates of the security of RC4. After several cypherpunk folks demonstrated how easy it is to cryptanalyze RC4 with the idle cycles of general-purpose computers, Netscape had to respond. Their note made several good points—for instance, that export controls were to blame, leaving them no choice but to use weak encryption—but their estimate of the cost of breaking 40-bit RC4 was greatly flawed. The first successful keysearch used idle cycles on 120 workstations for 8 days. Netscape claimed that this was $10,000 worth of computing power, concluding that messages worth less than $10,000 can be safely protected with 40-bit RC4 encryption [9]. Exposing the invalidity of this estimate was another motivating force for us.

One unpublished work [22] has studied in depth the relevance of reconfigurable logic to cryptologic applications. They assessed the complexity of a keysearch of DES and RC4 (as well as many other non-cryptanalytic problems). The main weakness of this aspect of their survey is that several of the estimates relied on theoretical predictions instead of real implementations and experimental measurements. In this paper, we attempt to give more rigorous estimates, paying attention to the architectural and economic issues facing these cryptanalytic applications.

**Technical Approach**

**Workloads and Architectures**

As we have explained earlier, there is much interest in the security of cryptographic algorithms. The algorithms with short keys (such as A5, RC4, CDMF, and DES) are the most interesting to examine, as their security depends intimately on the state-of-the-art in high-performance computing. Therefore, we concentrate on algorithms to break A5, RC4, CDMF, and DES.

Software implementations running on general-purpose microcomputers have received perhaps the most attention [3],[12],[5]. To achieve maximum performance, though, we must also consider the tradeoffs associated with customizable hardware. We will focus mainly on hardware implementations of cryptanalytic algorithms; we then compare the tradeoffs between the hardware and software approaches.

The most specialized approach involves using ASICs: custom-designed hardware, specially tailored to one particular cryptanalytic application. They require a significant initial investment for design and testing; they also must be produced in mass
quantity for them to be economical. Therefore, while probably the most efficient approach for a dedicated cryptanalytic application, ASICs require such a large investment that they are probably only of interest to small governments or large corporations—they are certainly not within reach for a class project!

Fortunately, there is a middle ground between ASICs and software. CPLDs (Complex Programmable Logic Devices) provide reconfigurable logic; they are commercially available at low prices. They provide the performance benefits of customizable hardware in small volume at a more reasonable price. We obtained access to a set of Altera FLEX8000 series programmable logic devices—more specifically, 81188GC232 chips.* These are mounted on a RIPP10 board, which can accommodate up to eight FLEX8000 chips and four 128KB SRAM memory chips.

Therefore, the primary platform of interest was the RIPP10 board with FLEX8000 chips; for comparison purposes, we also investigated several other programmable logic devices, as well as software-driven implementations. The workload consisted of brute-force cryptanalytic applications for RC4, A5, DES, and CDMF.

**The Figure of Merit**

It is important to keep in mind what quantities we are trying to measure. Regardless of whether the methodology involves real implementations or synthetic simulations, the ultimate figure of merit is the performance-cost ratio.

Why is the performance-cost ratio the relevant quantity? In general, our cryptanalytic applications are characterized by extreme suitability to parallelization: the process of exhaustive search over many keys can be broken into many independent small computations without penalty. One fast machine will finish the computation in exactly the same time as two machines which are twice as slow. Therefore, the relevant criterion is the “bang-to-buck” ratio, or more precisely, the numbers of trial keys searched per second per dollar.

**Methodology**

We used several methods to understand the architectural tradeoffs and their effect on cryptanalytic applications. We first implemented a few sample cryptanalytic algorithms and directly measured their performance on real workloads and actual architectures. Direct measurement is obviously the most desirable experimental technique; unfortunately, we do not have access to every system in existence. Therefore, to forecast the behavior on other platforms, we also used several simulation tools. In both cases, we examine actual applications and real systems.

* We greatly appreciate the kind support of Bruce Koball and Eric Hughes!
Direct measurement

Doing direct measurements on real systems running real applications is conceptually straightforward (but still labor-intensive in practice!). First, we directly implemented the relevant cryptanalytic algorithms for the Altera FLEX8000 platform. Once this is done, it is easy to do several small time trials to measure performance. Finally, we used technical data sheets [8] and price lists [7],[24] from Altera to assess the cost of the system.

We also implemented the applications in software. Measuring performance is easy; fixing a price on the computation is a bit less straightforward, and we will address that in a later section.

Simulations

It would be valuable to obtain measurements for a variety of CPLD architectures. As we only have access to the Altera RIPP10 board and FLEX8000 81188GC232 chips, the experimental procedure becomes a bit more involved. Fortunately, our development environment offers compilation, simulation, and timing analysis tools for several programmable logic devices. We therefore compiled the applications for several other chips and calculated predicted performance estimates with the simulation tools.

An important step for any simulation technique is to validate the simulation process. Accordingly, we applied the same simulation and timing analysis procedure to our applications for the FLEX8000 81188GC232; comparing the performance estimates from the simulation with the direct measurements lets us validate our experimental methodology.

Design and Analysis

Overview

We begin by setting up a model for analysis and describing several design issues that are common to all cryptanalytic hardware.

For this project, we are assuming the "known plaintext" model of cryptanalysis. In this model, an adversary has an encrypted message (the ciphertext), and also a small amount of the original message (the known plaintext). He also knows what part of the ciphertext corresponds to the known plaintext. The goal of the adversary is to determine the key necessary to decrypt the ciphertext into the known plaintext. He can then use this key to decrypt the rest of the encrypted message.

Other models of cryptanalysis, such as "ciphertext only" or "probabilistic plaintext" [29] are more complicated to use, but do not require an adversary to have specific
knowledge of part of the original message. However, as most messages have some well-known parts (a From header in a mail message, for example), the known plaintext model turns out to be applicable to almost all situations.

For a cryptographic algorithm to be considered secure, there must be no way to determine the decryption key which is faster than just trying every possible key, and seeing which one works (note that this is a necessary, but not sufficient, condition). This method is called brute force.

Breaking a cryptographic algorithm by brute force involves the following steps:

For each key in the keyspace
- Perform key setup
- Decrypt the ciphertext and compare it to the known plaintext

As will be seen below, different algorithms spend different amounts of time in the two steps. (For instance, stream ciphers—which generate output one bit at a time—allow us to prune incorrect key guesses very rapidly—while block ciphers—which operate on a block at a time—require us to generate the entire output block before any comparison is possible. DES and CDMF are block ciphers; A5 and RC4 are stream ciphers.)

We measure the expected number of cycles for each of the two steps for each key, and add them to determine a Cycles per Key, or CPK value for the algorithm.

Similar to the formula for CPU time found in [20]:

\[
\text{CPU time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock cycle time}
\]

we have a formula for brute-force searching a keyspace:

\[
\text{Search time} = \text{Keys to check} \times \text{CPK} \times \text{Clock cycle time}
\]

As with the [20] equation, we ignore CPU time. This is valid because we take care to avoid I/O as much as possible. Cryptanalytic applications are typically compute-bound, so this is an important optimization.

In the above formula, “Keys to check” indicates the number of keys to search; this can simply be the total number of keys that can be used with the algorithm, or, in the event that many chips are being used to simultaneously search the keyspace, it can be some fraction thereof.

“CPK”, as described above, is defined to be “KeySetup + Comparison”. “KeySetup” is the number of cycles required to load a key into the algorithm’s internal data structures, so that the key search engine is ready to produce output. “Comparison” is the expected number of cycles required for the algorithm to produce enough output so that it can be determined whether the key is the correct one. Note that
different algorithms divide their time differently between these two parts, as will be seen in more detail below.

"Clock cycle time" is exactly what one would expect; algorithms that attempt to do more complicated work in one cycle will tend to have a higher clock cycle time. This is also the factor that will vary most when using different models of hardware, as faster (more expensive?) chips have smaller gate delays. One important design feature common to all brute-forcing algorithms also affects this factor: how does one cycle through all of the keys in the keyspace? The obvious solution (to simply start at 0, and increment until the correct key is found) turns out to be a bad one, as incrementing a number of even 8 bits causes unacceptably large gate delays in propagating the carry. Tricks such as carry-save arithmetic \[20\] are usually not useful here, because keys are usually not used by the encryption algorithms as numbers, but rather, as bit strings.

A better solution \[31\], which uses the fact that the keys need not be checked in sequential order, is to use a linear feedback shift register \[27\], or LFSR. An LFSR is a register that can either be loaded (to set the register's value), or have its existing value shifted (in order to output 1 bit, and to change the register's value). Of the two styles of LFSR, the usual style is called a Fibonacci LFSR. To shift a Fibonacci LFSR, simply copy each bit to its neighbor on the right. The original rightmost bit is considered the output. The bit that is shifted in at the left is the parity of some specific subset of the bits (the taps) of the register (see Figure 10–1).

![Figure 10–1: Fibonacci LFSR](image)

The most important properties of an LFSR are that it has a low (constant) gate delay, and more importantly, if the taps are chosen properly, repeated shifting (starting with any non-zero value) will cycle through every possible non-zero value of the register.

The other style of LFSR is called a Galois LFSR, which has the same properties as the Fibonacci LFSR, but is shifted differently. To shift a Galois LFSR, copy each bit to its neighbor on the right, except for the taps, for which the rightmost bit of the register is XOR'd in before the copy is done. The bit that is shifted in at the left is the original rightmost bit, which is also considered the output (see Figure 10–2). The advantage of a Galois LFSR over a Fibonacci LFSR when being implemented in hardware is that a Galois LFSR usually has an even lower gate delay than a
Fibonacci LFSR, resulting in a potentially lower clock cycle time. For this reason, Galois LFSRs are usually used to cycle through the list of possible keys.

In order to take advantage of parallelism, one must be able to distribute the keyspace equitably among the multiple hardware devices. Standard mathematical techniques allow us to easily calculate the value of the shift register after any given number of shifts. From this, we can determine evenly separated starting positions for each device in the search engine.

We will now describe the design issues and analysis that were performed when we implemented various encryption algorithms in programmable logic.

**A5**

A5 [1] is the encryption algorithm used in GSM, the European standard for digital cellular telephones. It consists of three Fibonacci LFSRs of sizes 19, 22, and 23 respectively, which are initially loaded with the contents of the 64-bit key. The middle bits of all three LFSRs are examined at each clock cycle to determine which registers shift and which do not (at least two of the three registers shift in each clock cycle). The parity of the high bits of the LFSRs is output after each shift, and this output bitstream is XOR'd with the ciphertext to recover the original message.

This algorithm is quite well-suited for implementation in hardware due to the simplicity of LFSRs; given that it was designed for use in cellular phones, in which limited resources are available, this should not be surprising. The simplicity of the algorithm leaves almost no room for creativity to the implementer.

The resource requirements for A5 are quite minimal; they consist mainly of the 64 flipflops that make up the three LFSRs. In this algorithm, the key setup time is trivial (a single cycle to load the LFSRs with their initial state); the majority of the algorithm consists of comparing the output of the generator (which comes out at a rate of 1 bit per cycle) to the expected output. Since incorrect keys produce essentially random data, the expected number of bits we need to check before rejecting a key is 2. Thus, the total number of cycles per key for A5 is \( CPK = \text{KeySetup} + \text{Comparison} = 1 + 2 = 3 \).
RC4

RC4 [27] is the encryption algorithm used in, among other things, the Secure Sockets Layer (SSL) protocol [17] used by Netscape and other World Wide Web browsers to transmit encrypted information (such as banking transactions) over the Internet. RC4 is quite a simple algorithm; start with a 256-byte read-only array \( K \) that stores the key (repeat the key as often as necessary to fill \( K \)), a 256-byte random-access array \( S \), and two 8-bit registers \( i \) and \( j \).

To do key setup, start with \( j = 0 \), and do:

\[
\begin{align*}
\text{for } i &= 0 \text{ to } 255: \\
S[i] &= 1 \\
\text{for } i &= 0 \text{ to } 255: \\
&j = (j + S[i] + K[i]) \mod 256 \\
&\text{swap } S[i] \text{ and } S[j]
\end{align*}
\]

Once the key setup is complete, set \( i = j = 0 \), and to generate each byte, do:

\[
\begin{align*}
&i = (i + 1) \mod 256 \\
&j = (j + S[i]) \mod 256 \\
&\text{swap } S[i] \text{ and } S[j] \\
&\text{output } S[(S[i] + S[j]) \mod 256]
\end{align*}
\]

The sequence of bytes outputted is XOR'd with the ciphertext to recover the original message.

SSL, one common system that uses RC4, has a small added complexity. Instead of the key being copied into the array \( K \), as described above, it is first processed by the MD5 hash function; the result of the MD5 computation is then copied into \( K \). Our design and analysis does not include MD5, which is quite large, complicated, and includes many 32-bit additions, so readers hoping to break SSL should keep in mind that their performance will be substantially worse than that determined below.

The resource requirements for RC4 are considerable. Most notably, it requires 258 bytes of state (compare 8 bytes of state for A5), 256 bytes of which need to be accessed randomly. Such resources were beyond the capabilities of the programmable logic chips we had available, but fortunately the board on which the logic chips were mounted had 128KB of SRAM accessible to the logic chips via a bus; we stored the array \( S \) in this SRAM. Note that the key array \( K \) is accessed in a predictable order, so it was not necessary to store it in the SRAM.

Unfortunately, when trying to produce instruction-level parallelism in the algorithm, the single port to the SRAM becomes a structural hazard. For this reason, it was necessary to serialize accesses to this SRAM. Initially, we expected that going off-chip to access the SRAM would be the bottleneck that determined the mini-
mom clock cycle time; the section entitled “Analysis” below shows that we were incorrect.

We now calculate the “Cycles per Key” value for RC4. Examining the key setup code, it is clear that the first loop requires 1 cycle to initialize $i$ to 0, and 256 cycles to complete, and each iteration of the second loop requires 4 cycles (1 each to read and write $S[i]$ and $S[j]$), for a total key setup time of 1281 cycles.

Similarly, each byte of output requires 5 cycles to produce (1 each to read and write $S[i]$ and $S[j]$, and 1 to read $S[(S[i] + S[j]) \mod 256]$). The expected number of bytes needed to determine whether the guessed key is correct is:

$$(1 - \frac{1}{256})^{-1} < 1.004$$

so the value of “Comparison” is very near 5. Thus we calculate the total Cycles per Key to be $\text{CPK} = \text{KeySetup} + \text{Comparison} = 1281 + 5 = 1286$.

**DES and CDMF**

DES is the national Data Encryption Standard; it enjoys widespread use by the banking industry, as well as being one of the preferred algorithms for securing electronic communications. DES transforms a 64 bit input block into a 64 bit output by a reversible function which depends on the 56 bit key in a highly non-linear way.

The DES algorithm was designed primarily for efficiency in hardware, and thus has several distinguishing features worth noting. It consists of an initial and final permutation and 16 rounds of main processing, with each round transforming the input bits via a “mix-and-mash” process. Bit permutations are used extensively; of course, they are trivial to do in hardware by simply reordering wires. Each round also contains 8 different “Substitution” boxes (or S-boxes for short); the S-boxes are non-linear functions which map 6 input bits to 4 output bits. S-boxes are not very resource-intensive in hardware: they can be implemented as four 6-input boolean functions, and their small size keeps the gate count reasonable. The key is stored in a shift register, rotated before each round, and exclusive-or-ed into the block during each round. This is also straightforward to implement in hardware.

CDMF (Commercial Data Masking Facility) [23] is a related algorithm which uses DES as the underlying transformation; the only difference is that it weakens the key to meet US export restrictions. CDMF has an effective 40-bit keylength, which is then expanded to a 56 bit DES key by using another DES transformation. Loading a CDMF key requires one initial DES operation, and transforming each 64 bit block requires one DES operation. Therefore an implementation of a DES key-
search application leads easily to a CDMF keysearch engine with half the search rate.

Our DES implementation was forced to be rather minimal to fit in the limited resources available on our chip. We implemented one round of DES, with the appropriate S-boxes and bit permutations. Some extra flip-flops and a state machine allow us to iterate the round function 16 times; there was not sufficient space (i.e. logic gates) available to implement 16 instantiations of each S-box.

The S-boxes are perhaps the most critical component, and we tried several different implementation approaches for them. One natural way to describe each S-box is as a 64-entry lookup table containing 4 bit entries. This might be a good choice if the chip had contained some user-configurable ROM; ours didn’t. A similar approach takes advantage of the compiler support for “case” statements, which gets translated into a hardware structure containing a 64-line demultiplexor and or gates expressing the relevant minterms. This structure minimizes gate delay at the expense of space resources. In fact, this structure increased the gate requirements significantly, to the point where the 8 S-boxes alone required more hardware resources than our overworked chip had to offer. The compiler was not particularly helpful at doing space-time tradeoffs to minimize the space requirements, so we ended up optimizing the S-box functions by hand.

The manual optimization we settled on can be viewed as a form of speculative execution. First, note that it suffices to describe how to compute the 6-bit to 1-bit boolean function that calculates one output bit of some S-box. Since the S-boxes behave roughly like they were chosen at random, we don’t expect to find any structure in the outputs—i.e. each output will be an uncorrelated non-linear function of the inputs—so this is roughly optimal. To compute such a 6-to-1 function, we first isolate 2 of the 6 input bits as control bits. We do speculative execution with four functional cells; each cell computes the output of the 6-to-1 function under a speculative assumption about the 2 control bits. As there are four possible values of the control bits, the four functional cells enumerate all possibilities. At the same time the functional cells are computing their 4-to-1 function, a multiplexor unit concurrently selects one of the functional cells. The calculation of the 6-to-1 function via speculative execution is depicted in Figure 10-3. This choice of S-box implementation structure is tailored to our Altera FLEX8000 chips: these chips are organized as an array of logic cells, where each logic cell can compute an arbitrary (configurable) 4-to-1 boolean function. For chips with a different organization, some other manual optimization might be more appropriate.

The “Search time” equation for our CDMF implementation is not hard to analyze. One can easily count the CPK by direct inspection of our implementation. We have a finite state machine with 4 states, labelled from a to d. The cycle-by-cycle breakdown of the “KeySetup” time for one CDMF encryption is as follows:
Figure 10-3: Calculation of a boolean function with 6 inputs

a. 1 cycle to increment the key and load in the 40-bit CDMF trial key
b. 1 cycle to perform the DES input permutation
c. 16 cycles to perform 16 rounds of encryption
d. 1 cycle to perform the DES final permutation and load in the 64 bit plaintext block

We can see that the “KeySetup” time is 19 cycles. An enumeration of the output generation and comparison stage yields

a. 1 cycle to perform the DES input permutation
b. 16 cycles to perform 16 rounds of encryption
c. 1 cycle to perform the DES final permutation, compare the ciphertext block to the expected value, and return to state a if this trial key was incorrect

This means that the “Comparison” time is 18 cycles, so the total CPK is $19 + 18 = 37$. Note that DES encrypts the entire 64 bit block at once, unlike a stream cipher, so we check all of the output bits in parallel.

The hardware resources required by CDMF are reasonable but non-negligible for commercial CPLDs. Our minimal implementation required (the equivalent of) roughly 10000 gates. This is certainly within reach for many newer commercial CPLDs, although there are also many older or less expensive CPLDs which cannot handle the requirements. It is important to keep the entire keysearch engine on one chip; otherwise, inter-chip I/O will severely limit performance.

**Analysis**

We cross-compiled our cryptanalysis implementations for many different Altera CPLDs, and ran a simulation and timing analysis to measure the maximum applicable clock cycle time. The results are plotted in Figure 10–4 for CDMF, Figure 10–5 for A5, and Figure 10–6 for RC4. Some explanation is in order, as there are a lot of data summarized there. The chip specification (e.g. 81188GC232-3) can be dissected as follows: the 81188 refers to the general family, the 232 specifies a 232-pin package, and the -3 refers to the speed grade (lower numbers are faster). The 81500 is the top of the line Altera FLEX8000 device; the 81188 is a bit less powerful. Chips without the “A” designation were fabricated with an older .8 micron process; the “A” indicates chips that were manufactured with a newer, faster .6 micron process. The figure shows throughput graphed against the initial investment required; the chips with the best performance-to-cost (Y/X) ratio are the best buy. The prices are taken from a very recent Altera price list [7],[24]. As there are discounts in large quantities, we have plotted price points for small quantities with a red line and for large batches with a blue line.

We also measured the performance for the 81188GC232-3 chip directly—it is the only one we had access to. Our measurements agreed closely with the simulated timing analysis, confirming the validity of our experimental methodology.
Measurements for DES are not listed. Nonetheless, they track the CDMF performance figures very closely. CDMF consists of two DES encryptions—one for key setup, and one for output generation—with very little overhead. The DES keysearch rates can be derived from Figure 10-4 by simply doubling the CDMF rate. Also, remember that the DES keyspace is $2^{16}$ times as large. Our data indicate that if one wanted a machine which could perform a DES keysearch in a year on average, it would suffice to spend $45,000 to buy 600 of the Altera 81500ARC240-4 CPLDs. (This is a very rough estimate, which does not include overhead such as mounting shelves, etc.)

![Figure 10-4: CDMF cryptanalysis economics](image)

One can note several interesting things from the graph. First, examine the peculiar zig-zag nature of the 81188ARC240 lines. The points are plotted in order of the chip's rated speed grade, from A-6 on the bottom to A-2 on the top. The strange "zag" occurs because the price for a faster A-4 chip drops significantly below the price for the slower A-5. Altera specifies the A-4, A-3, and A-2 as their "preferred" grades for that chip, presumably because there is more sales volume for those speed grades. If you were to build a keysearch engine out of 81188ARC240 chips, you should try to be right at the "hump"—the A-4 speed grade is the best buy for that chip.
We have not yet explained the two leftmost dotted lines. The 81500 line of chips contains more hardware resources than the 81188—1296 instead of 1008 “logic elements”—and this extra space should be taken into account when comparing hardware devices. With our A5 and CDMF implementations, there is quite a bit of space left over on the 81500 chip, as it turns out. Therefore, it is natural to ask whether two independent key trial engines might fit on the same chip. We believe (from close examination of the resource usage) that, with A5 and CDMF, there are sufficient hardware resources on the 81500 to support two superscalar keysearch operations. (It would admittedly be a tight fit.) Because of time pressures, we have not actually implemented this. RC4 requires, it seems, too many resources (mainly flip-flops for internal state) to use this strategy. There would be other difficulties with RC4, anyhow—one would probably need a dual-ported SRAM, or two SRAM chips attached to the CPLD (as discussed below).

One might wonder why we proposed taking advantage of extra hardware resources with a multiple-issue architecture, instead of using (say) advanced pipelining techniques. It is worthwhile to recall why advanced pipelining techniques were developed. On a traditional general-purpose computer, programs are typically serialized so highly that if one were to implement several independent simple processors on the same chip, there simply would not be enough tasks to
keep the co-processors busy with useful work. Architects have been blessed with plentiful hardware resources and cursed with the need to speed up single-instruction-stream uniprocessors; this explains the proliferation of sophisticated pipelining methods. (Of course, pipelining does not provide linear speedup with linear increases in hardware resources, like parallelism would, but it is better than nothing!) We are faced with an entirely different situation here. Our cryptanalytic applications encourage virtually unlimited parallelism, so there is no need to look to sophisticated caching schemes for speeds. Achieving parallelism via a superscalar architecture is both simpler and more effective for our purposes.

The projected performance for parallelized 81500 A5 and CDMF keysearch is indicated on the plots with a green and block dotted line, labelled “half_81500ARC240 family”, with the unit price halved to indicate its factor-of-two multiple-issue nature. (We could have doubled the performance instead, but that would have made the graph harder to read, so for ease of comprehension and comparison we chose to halve the cost instead.)

We discussed in class why the future of high-performance computing lies in massively-parallel collections of low-end processors (say, Pentiums), instead of in specialized advanced CPUs. One major reason is that Pentium processors are sold in such large quantities that tremendous economies of scale apply, and specialized
processors simply cannot compete with the low-end's ever-increasing performance-cost ratio. We can see that an analogous situation applies here as well. The graphs show that, for our applications, upgrading to a higher speed grade is almost never worth the increased cost. (Two notable exceptions—the "hump" in the 81188ARC240 plot, and the benefits of using a 81500 with enough hardware resources to implement two keysearch engines on-chip—have already been discussed.) Within each family, the least expensive chip turns out to yield the best performance-to-cost ratio; spending twice as much money on a higher-grade chip in the family never results in twice the performance. On the other hand, upgrading to a more recent “A” designated family—one fabricated with a newer .6 micron process—is a worthwhile move. Altera has listed the “A” chips as their preferred technology, and presumably there is more sales volume for devices on their preferred list (though it might be hard to separate cause from effect here). These charts don’t tell the whole story. Altera is as we write starting to release a new advanced line of reconfigurable logic devices, the FLEX10K architecture. In recommending the 81188 and 81500 devices, we gain extra price-performance benefits by staying a bit behind behind the bleeding edge. Exploiting parallelism with low-end devices is a win for our applications.

We have not yet discussed the impact of software in relation to the hardware performance measurements. Software is a bit trickier to evaluate and compare to the other measurements, as it is not clear how to compare the price of a software solution to a hardware approach. While hardware devices would typically be purchased with one application in mind, often a certain amount of idle cycles on general-purpose computers is available “for free”. Nonetheless, software and hardware approaches typically won’t be in serious competition: the extra expense of hardware is usually not justified until “free” software implementations on general-purpose computers are unacceptably slow.

Table 10–1: Typical software performance on cryptanalytic applications

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Keys searched per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC4</td>
<td>21900</td>
</tr>
<tr>
<td>CDMF</td>
<td>29800</td>
</tr>
<tr>
<td>DES</td>
<td>41300</td>
</tr>
<tr>
<td>A5</td>
<td>355000</td>
</tr>
</tbody>
</table>

Table 10-1 lists the performance of brute-force keysearch applications, as measured on a Pentium P100 machine. Of course these figures will vary widely from computer to computer. For example, we estimate that we could perform a distributed RC4 40-bit keysearch in a weekend or so, and a CDMF 40-bit keysearch in about a night or two, by using idle cycles on the hundreds of general-purpose computers we have access to as Berkeley computer science graduate students.
Many other organizations also have large numbers of computers which are idle much of the time. Many employees and students thus have access to spare computational power which may be harnessed for cryptanalysis, at essentially zero cost. Compare this to Netscape's estimate that amassing enough processing power to break 40-bit RC4 would cost roughly $10,000. For much less than this, one could probably convince a starving graduate student to lend out access to the necessary computer account. In any event, if Netscape were willing to pay $10,000 for the amount of computing power required to break 40-bit RC4, some enterprising student could easily form a extremely profitable business model.

Given a distributed system of general-purpose computers, one can easily compute the maximum rate of 40-bit keysearching possible in idle cycles by assuming that most machines are idle at least half of the time and using estimates such as those in Table 10–1; achieving better performance than this calls for hardware. We can see from Table 10–1 that our hardware implementations of CDMF, DES, and A5 keysearch are orders of magnitude faster than software; this is not surprising, as these encryption algorithms were designed for efficiency in hardware.

RC4, by contrast, was designed to run efficiently in software, and indeed, as can be seen by comparing Figure 10–6 and Table 10–1, RC4 performs about twice as well in software than on programmable logic. The primary reasons for the large search time on programmable logic are that RC4 has a large "Cycles per Key" value, and a large "Clock cycle time" value: as seen above, the total CPK for the RC4 algorithm is 1286; far larger than the 3 for A5 or the 37 for CDMF. The large clock cycle time stems from the fact that the algorithm contains a number of register additions; as discussed above, these can produce very large gate delays. Unfortunately, changing the additions to LFSRs (as was done above), or using tricks such as carry-save arithmetic, is not appropriate for RC4, as can be seen by examining the algorithm.

Another blow to implementing RC4 efficiently was the particular hardware architecture we had. The programmable logic devices we used were not large enough to store the necessary 256-byte state array on-chip, so we were forced to store them in the external SRAM. However, the algorithm utilizes the SRAM every cycle, so the number of simultaneous RC4 trials we can compute is limited by the number of ports to SRAM that we have available. Unfortunately, on the RIPP10 programming board, not only is the SRAM single-ported, but each SRAM is shared by two logic chips. Thus on a fully-populated board with eight logic chips and four SRAMs, we can only perform four simultaneous RC4 trials. Redesigning the programming board to include a port to SRAM for each simultaneous RC4 trial would save some overhead (wasted space on the board), but would not increase the relatively poor performance to cost ratio shown above.
One advantage of software is that the development process is significantly easier. By reusing code (from cryptographic libraries available on the Internet, for example), we prototyped RC4, A5, CDMF, and DES software keysearch applications in a total time of under an hour. In contrast, our programmable logic design and implementation effort took roughly 4 weeks to complete.

Programmable logic has similar advantages over custom-hardware. Development and design would be still more time-consuming and costly for a custom-hardware approach, such as an ASIC. Furthermore, such an ASIC can only be used for one limited algorithm. Programmable logic is more flexible—the hardware devices can be reused for cryptanalysis of many different encryption algorithms with little extra effort. Apparently AccessData, a business that specializes in recovering lost data (i.e. cryptanalysis) for the corporate and law enforcement industries, prefers programmable logic over custom hardware for exactly these reasons [28].

Let us summarize what the charts recommend to one in need of cryptanalytic computational power. RC4 keysearches appear to be most efficiently performed in general-purpose distributed systems. Performing a single isolated 40-bit CDMF keysearch is perhaps best done with distributed software, if time is not of the essence and there are sufficient general-purpose computational resources easily available. For CDMF and A5 keysearch in anything more than that extremely minimal setting, though, reconfigurable logic is the most appropriate solution of the technologies that we examined. Of the devices we surveyed, the Altera 81500ARC240-4 device is the most appropriate and economical choice for cryptanalytic applications; for instance, a $15,000 initial investment buys about 200 of these chips, allowing one to perform on average one CDMF keysearch every hour. The cost scales linearly, requiring approximately $10^8$ dollar-seconds for a complete CDMF keysearch; that is, an initial investment of X dollars allows one to search the entire CDMF keyspace in $10^8X$ seconds, while the average time to find a key is half that. In addition, we provisionally estimate that about $45,000 of CPLD hardware could perform a DES keysearch in a year, as calculated above. Table 10-2 summarizes some of these calculations. It takes into account the economies of scale associated with buying many logic devices, and is based on the average-case (not worst-case) search time; the worst-case figure would be twice as large. No figures for A5 are included, because at the moment, there is no consensus among cryptographers as to the size of the keyspace [26].
Table 10-2: Estimating the cost of cryptanalysis: a summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Investment for average keysearch time of</th>
<th>Architecture components</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 year</td>
<td>1 week</td>
</tr>
<tr>
<td>RC4</td>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>CDMF</td>
<td>$0</td>
<td>$0</td>
</tr>
<tr>
<td>CDMF</td>
<td>$93</td>
<td>$93</td>
</tr>
<tr>
<td>DES</td>
<td>$45,000</td>
<td>-</td>
</tr>
</tbody>
</table>

Future work

Due to time and resource limitations, we were only able to examine the Altera FLEX8000 series of programmable logic devices. An obvious extension of this work would be to examine other kinds of devices, such as the new Altera FLEX10K series, or devices from other vendors such as Xylinx. Additionally, it would be worthwhile to examine the technology trends in programmable logic, to determine how they compare to those for general-purpose hardware.

We leave it as an open problem to the reader to actually construct a fully operational DES keysearch engine.

Conclusions

We found that RC4 cryptanalysis is most effectively implemented in software. Since RC4 was specifically designed for efficiency on general-purpose computers, it is not entirely surprising that programmable logic fares so poorly. We showed that the estimate in [4] (which inspired the term “8-cent encryption” for 40-bit RC4) is over-optimistic and unrealistic. On the other hand, Netscape’s $10,000 estimate was far too large.

Programmable logic devices are very efficient at CDMF cryptanalysis. We estimate that an initial investment of $745 buys enough programmable logic to recover one CDMF key each day; this shows that CDMF is practical to break. Moreover, DES is nearly practical to break; a cryptanalytic engine to do a DES keysearch each year can be built with roughly $45,000 of programmable logic.

Several architectural techniques from the design of general-purpose processors were useful in this project. Adding parallelism, identifying structural and data hazards, identifying performance bottlenecks, and other techniques helped maximize the performance of our design. The cryptanalytic analogue to the “CPU time” equation from [20] was surprisingly useful, lending structure to our analysis.

We also identified several important aspects found only with cryptanalytic applications on programmable logic. In this application, superscalar parallelism is more
effective than pipelining. Also, register additions can often be a limiting bottleneck for programmable logic—we avoided them where possible, and suffered large performance hits elsewhere.

By considering architectural issues both common to general-purpose processors and unique to programmable logic, we examined the feasibility of using commodity logic devices for cryptanalytic applications.

Acknowledgements

This work would not have been possible without the assistance of a number of people. We would like to thank Eric Hughes and Bruce Koball for providing the hardware and software. We would also like to thank Clive McCarthy and Stephen Smith, both of Altera, for their generous support.

Availability

This paper, and other related materials, are available on the World Wide Web at http://www.cs.berkeley.edu/~iang/isaac/hardware/.

References

Post to sci.crypt newsgroup. Available on the Internet as http://chem.leeds.ac.uk/ICAMS/people/jon/a5.html,


Post to cypherpunks mailing list. Available on the Internet as http://dcs.ex.ac.uk/~aba/ssl/.

[4] Matt Blaze and Whitfield Diffie and Ronald L. Rivest and Bruce Schneier and Tsutomu Shimomura and Eric Thompson and Michael Wiener, Minimal key lengths for symmetric ciphers to provide adequate commercial security: A report by an ad hoc group of cryptographers and computer scientists, Business Software Alliance, January, 1996
Available on the Internet as http://www.bsa.org/policy/encryption/cryptographers.html

Available on the Internet as http://www.brute.cl.cam.ac.uk/brute/.

References

91, Springer-Verlag, 1992, pages 367-376


Available on the Internet as http://www.altera.com/

[9] Netscape Communications Corporation, Key Challenge, 1995

Post to cypherpunks mailing list. Available on the Internet as http://www.eskimo.com/~weidai/benchmarks.txt


[12] Damien Doligez, SSL challenge—broken, August, 1995
Post to cypherpunks mailing list. Available on the Internet as http://paul-lac.inria.fr/~doligez/ssl/,


Post to cypherpunks mailing list. Available on the Internet as http://www.rain.org/~hal/sslchallong.html

Internet-Draft draft-freier-ssl-version3-00.txt, work in progress


[24] Clive McCarthy, Personal communication, April, 1996


[26] Michael Roe, Personal communication, April, 1996


[28] Bruce Schneier, Personal communication, April, 1996


Efficient DES
Key Search—An Update
by Michael J. Wiener

An exciting moment in the history of DES was reached in June 1997 when a group coordinated by Rocke Verser solved RSA Data Security’s DES challenge by exhaustive key search on a large number of computers. This result was useful because it served to underscore in a public way how vulnerable DES has become. However, it may also have left the false impression that one cannot do much better than attacking DES in software with a large distributed effort. The design of DES is such that it is fairly slow in software, but is compact and fast when implemented in hardware. As a result, using software to attack DES gives poor performance compared to what can be achieved in hardware. This applies not only to DES, but also to most other block ciphers, attacks on hash functions, and attacks on elliptic curve cryptosystems. Avoiding efficient hardware-based attacks requires the use of algorithms with sufficiently long keys, such as triple-DES, 128-bit RC5,* and CAST-128.†

In this article we assess the cost of DES key search using hardware methods and examine the effectiveness of some proposed methods for thwarting attacks on DES.

Michael J. Wiener, Entrust Technologies, 750 Heron Road, Suite E08, Ottawa, Ontario, Canada K1V 1A7
This article first appeared in RSA Laboratories’ Autumn 1997 Cryptobytes newsletter; it is reprinted with permission from the author and RSA Data Security, Inc.

Advancing Technology

The best known way to attack DES is to simply try all of the possible 56-bit keys until the correct key is found. On average, one expects to go through about half of the key space. In 1993, a design for an exhaustive DES key search machine including a detailed chip design was published.* A $1 million version of this machine used 57600 key search chips, each capable of testing 50 million keys per second. Overall, the machine could find a DES key in, on average, three and a half hours.

About four and a half years have passed since this design was completed, and according to Moore's Law, processing speeds should have doubled three times in that period. Of course, estimating in this fashion is a poor substitute for the careful analysis and design effort that went into the earlier design. The original chip design was done in a 0.8 micron CMOS process, and with the geometries available today, it is possible to fit four instances of the original design into the same silicon area. In keeping with the conservative approach to estimates in the 1993 paper, we assume here that the updated key search chip's clock speed would increase to only 75 MHz from the original 50 MHz, making the modern version of the chip six times faster for the same cost. It is interesting to note that just 21 of these chips would give the same key searching power as the entire set of computers used by the team who solved the DES challenge.

Today's version of the $1 million machine could find a DES key in, on average, about 35 minutes (one-sixth of 3.5 hours). This time scales linearly with the amount of money spent as shown in the following table.

<table>
<thead>
<tr>
<th>Key Search Machine Cost</th>
<th>Expected Search Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10,000</td>
<td>2.5 days</td>
</tr>
<tr>
<td>$100,000</td>
<td>6 hours</td>
</tr>
<tr>
<td>$1,000,000</td>
<td>35 minutes</td>
</tr>
<tr>
<td>$10,000,000</td>
<td>3.5 minutes</td>
</tr>
</tbody>
</table>

Note that the costs listed in the table do not include the cost to design the chip and boards for the machine. Because the one-time costs could be as high as half a million dollars, it does not make much sense to build the cheaper versions of the machine, unless several are built for different customers.

This key search engine is designed to recover a DES key given a plaintext-cipher-text pair for the standard electronic-codebook (ECB) mode of DES. However, the machine can also handle the following modes without modification: cipher-block

chaining (CBC), 64-bit cipher feedback (CFB), and 64-bit output feedback (OFB). In the case of OFB, two consecutive plaintexts are needed. The chip design can be modified to handle two other popular modes of DES, 1-bit and 8-bit CFB, at the cost of a slightly more expensive chip. Fewer chips could be purchased for a $1 million machine causing the expected key search time to go up to 40 minutes for all modes, except 1-bit CFB, which would take 80 minutes, on average.

Programmable Hardware

The costs associated with chip design can present a significant barrier to small-time attackers and hobbyists. An alternative which has much lower start-up costs is the use of programmable hardware. One such type of technology is the Field Programmable Gate Array (FPGA). One can design a circuit on a PC and download it to a board holding FPGAs for execution. In a report in early 1996,* it was estimated that $50000 worth of FPGAs could recover a DES key in, on average, four months. This is considerably slower than what can be achieved with a chip design, but is much more accessible to those who are not well funded.

Another promising form of programmable hardware is the Complex Programmable Logic Device (CPLD). CPLDs offer less design freedom and tend to be cheaper than FPGAs, but the nature of key search designs seems to make them suitable for CPLDs. Further research is needed to assess whether CPLDs are useful for DES key search.

Avoiding Known Plaintext

The designs described to this point have relied on the attacker having some known plaintext. Usually, a single 8-byte block is sufficient. One method of preventing attacks that has been suggested is to avoid having any known plaintext. This can be quite difficult to achieve. Frequently, data begins with fixed headers. For example, each version of Microsoft Word seems to have a fixed string of bytes that each file begins with.

For those cases where a full block of known plaintext is not available, it is possible to adapt the key search design. Suppose that information about plaintext is available (e.g., ASCII character coding is used), but no full block is known. Then instead of repeatedly encrypting a known plaintext and comparing the result to a ciphertext, we repeatedly decrypt the ciphertext and test the candidate plaintexts against our expectations. In the example where we expect 7-bit ASCII plaintext, only about 1 in 256 keys will give a plaintext which has the correct form. These

---

keys would have to be tried on another ciphertext block. The added logic to handle this would add just 10 to 20% to the cost of a key search chip.

Even if we only know a single bit of redundancy in each block of plaintext, this is enough to cut the number of possible keys in half. About 56 such blocks are needed to uniquely identify the correct key. This does not mean that the run-time is 56 times greater than the known-plaintext case. On average, each key is eliminated with just two decryptions. Taking into account the cost of the added logic required makes the expected run-time for a $1 million machine about 2 hours in this case.

**Frequent Key Changes**

A commonly suggested way to avoid key search attacks is to change the DES key frequently. The assumption here is that the encrypted information is no longer useful after the key is changed, which is often an inappropriate assumption. If it takes 35 minutes to find a DES key, why not change keys every 5 minutes? The problem with this reasoning is that it does not take exactly 35 minutes to find a key. The actual time is uniformly distributed between 0 and 70 minutes. We could get lucky and find the key almost right away, or we could be unlucky and take nearly 70 minutes. The attacker’s probability of success in the 5-minute window is 5/70 = 1/14. If after each key change the attacker gives up and starts on the next key, we expect success after 14 key changes or 70 minutes. In general, frequent key changes cost the attacker just a factor of two in expected run-time, and are a poor substitute for simply using a strong encryption algorithm with longer keys.

**Conclusion**

Using current technology, a DES key can be recovered with a custom-designed $1 million machine in just 35 minutes. For attackers who lack the resources to design a chip and build such a machine, there are programmable forms of hardware such as FPGAs and CPLDs which can search the DES key space much faster than is possible using software on PCs and workstations. Attempts to thwart key search attacks by avoiding known plaintext and changing keys frequently are largely ineffective. The best course of action is to use a strong encryption algorithm with longer keys, such as triple-DES, 128-bit RC5, or CAST-128.
The Electronic Frontier Foundation

Electronic Frontier Foundation
1550 Bryant Street, Suite 725
San Francisco CA 94103 USA
+1 415 436 9333 (voice)
+1 415 436 9993 (fax)
http://www.eff.org
info@eff.org

The Electronic Frontier Foundation (EFF) is a nonprofit public-interest organization protecting rights and promoting liberty online. It was founded in 1990 by Mitchell Kapor, John Perry Barlow, and John Gilmore.

The Foundation seeks to educate individuals, organizations, companies, and governments about the issues that arise when computer and communications technologies change the world out from under the existing legal and social matrix.

The Foundation has been working on cryptography policy for many years. It was a significant force in preventing the adoption of the “Clipper chip” and its follow-on “key escrow” proposals, and continues to advocate for wide public availability and use of uncompromised and unbreakable encryption technology. EFF is backing the lawsuit in which Professor Daniel Bernstein seeks to overturn the United States export laws and regulations on cryptography, arguing that the First Amendment to the US Constitution protects his right to publish his cryptography research results online without first seeking government permission. EFF’s research effort in creating this first publicly announced DES Cracker, and the publication of its full technical details, are part of EFF’s ongoing campaign to understand, and educate
the public about, the social and technical implications of cryptographic technology.

EFF encourages you to join us in exploring how our society can best respond to today's rapid technological change. Please become an EFF member; see http://www.eff.org/join/.

**John Gilmore**

John Gilmore is an entrepreneur and civil libertarian. He was an early employee of Sun Microsystems, and co-founded Cygnus Solutions, the Electronic Frontier Foundation, the Cypherpunks, and the Internet's "alt" newsgroups. He has twenty-five years of experience in the computer industry, including programming, hardware and software design, and management. He is a significant contributor to the worldwide open sourceware (free software) development effort. His advocacy efforts on encryption policy aim to improve public understanding of this fundamental technology for privacy and accountability in open societies. He is currently a board member of Moniker pty ltd, the Internet Society, and the Electronic Frontier Foundation.

John leads the EFF's efforts on cryptography policy, managed the creation of the DES cracker, and wrote much of the text in this book.

John can be reached at the email address gnu@des.toad.com; his home page is http://www.cygnus.com/~gnu/.

**Cryptography Research**

Cryptography Research
870 Market Street, Suite 1088
San Francisco, CA 94102 USA
+1 415 397 0123 (voice)
+1 415 397 0127 (fax)
http://www.cryptography.com

Cryptography Research is Paul Kocher's San Francisco-based consulting company. Cryptography Research provides consulting, design, education, and analysis services to many leading firms and start-ups. Kocher and the company are widely known for their technical work and research, including the development of leading cryptographic protocols (such as SSL 3.0), cryptanalytic work (including the discovery of timing attacks against RSA and other cryptosystems), and numerous presentations at major conferences. To reach Cryptography Research please write to info@cryptography.com.
Cryptography Research managed the hardware and software design for the DES cracker, and wrote the chip simulator and the driver software.

Paul Kocher, Josh Jaffe, and everyone else at Cryptography Research would like to thank John Gilmore and the EFF for funding this unique project, and AWT for their expert hardware work!

**Paul Kocher**

Paul Kocher is a cryptographer specializing in the practical art of building secure systems using cryptography. He currently serves jointly as President of Cryptography Research (http://www.cryptography.com) and Chief Scientist of ValiCert (http://www.valicert.com). Paul has worked on numerous software and hardware projects and has designed, implemented, and broken many cryptosystems. Paul can be reached via e-mail at paul@cryptography.com.

**Advanced Wireless Technologies**

Advanced Wireless Technologies, Inc.
3375 Scott Blvd, Suite 410
Santa Clara, CA 95054 USA
+1 408 727 5780 (voice)
+1 408 727 8842 (fax)
http://www.awti.com

Advanced Wireless Technologies, Inc. (AWT) is dedicated to providing Application-Specific Integrated Circuit (ASIC) and board level design solutions for high tech industries at highest quality and lowest cost. AWT’s design philosophy is to reduce product development cost/risk and recurring cost. AWT employs a thorough design flow from system architecture to system integration and test.

AWT was founded in 1993. Its engineering team is composed of a highly qualified, tenured employee base, including technical management staff. The employees are knowledgeable, motivated, highly competent, and have from 3 to 25 years of experience in system engineering, chip design, and complete subsystem design.

AWT offers digital ASIC/Gate Array and Board design services to support customers’ specific requirements. The company can participate in any development phase from specifications definition to design implementation and prototype testing.

In addition to providing engineering services AWT has developed leading products for use in the communications industry. AWT’s standard products include IP Cores,
ASICs, and board level products in the fields of demodulation, forward error correction, and encryption/decryption.

AWT designed and built the hardware for the DES Cracker, including the custom ASIC, logic boards, and interface adapters. If you're interested in purchasing a DES Cracker unit, contact AWT.

AWT invites you to visit at http://www.awti.com or call +1 408 727 5780 for your specific engineering needs.
**Titles from O’Reilly**

**Web Programming**
- Advanced Perl Programming
- Apache: The Definitive Guide
- Building Your Own Web Conferences
- Building Your Own Website™
- CGI Programming
  - for the World Wide Web
  - Designing for the Web
- Dynamic HTML:
  - The Complete Reference
- Frontier: The Definitive Guide
- HTML: The Definitive Guide,
  - 2nd Edition
- Information Architecture
  - for the World Wide Web
- JavaScript: The Definative Guide,
  - 2nd Edition
- Learning Perl,
  - 2nd Edition
- Learning Perl for Win32 Systems
- Mastering Regular Expressions
- Netscape IFC in a Nutshell
- Perl5 Desktop Reference
- Perl Cookbook
- Perl in a Nutshell
- Perl Resource Kit—UNIX Edition
- Perl Resource Kit—Win32 Edition
- Programming Perl,
  - 2nd Edition
- WebMaster in a Nutshell,
  - Deluxe Edition
- Web Security & Commerce
- Web Client Programming with Perl

**Graphic Design**
- Director in a Nutshell
- Photoshop in a Nutshell
- QuarkXPress in a Nutshell

**Java Series**
- Database Programming with
  - JDBC and Java
- Developing Java Beans
- Exploring Java,
  - 2nd Edition
- Java AWT Reference
- Java Cryptography
- Java Distributed Computing
- Java Examples in a Nutshell
- Java Fundamental Classes Reference
- Java in a Nutshell,
  - 2nd Edition
- Java in a Nutshell,
  - Deluxe Edition
- Java Language Reference,
  - 2nd Edition
- Java Native Methods
- Java Network Programming
- Java Security
- Java Threads
- Java Virtual Machine

**Songline Guides**
- NetLaw
- NetResearch
- NetLearning
- NetSuccess
- NetLessons
- NetTravel

**System Administration**
- Building Internet Firewalls
- Computer Crime:
  - A Crimefighter’s Handbook
- Computer Security Basics
- DNS and BIND,
  - 2nd Edition
- Essential System Administration,
  - 2nd Edition
- Essential Windows/NT System Administration
- Getting Connected:
  - The Internet at 56K and Up
- Linux Network Administrator’s Guide
- Managing Internet Information Services,
  - 2nd Edition
- Managing IP Networks
  - with Cisco Routers
- Managing Mailing Lists
- Managing NIS and NIS
- Managing the WinNT Registry
- Managing Usenet
- MCSE: The Core Exams in a Nutshell
- MCSE: The Electives in a Nutshell
- Networking Personal Computers
  - with TCP/IP
- Palm Pilot: The Ultimate Guide
- Practical UNIX & Internet Security,
  - 2nd Edition
- PGP: Pretty Good Privacy
- Protecting Networks with SATAN
- sendmail,
  - 2nd Edition
- sendmail Desktop Reference
- System Performance Tuning
- TCP/IP Network Administration,
  - 2nd Edition
termcap & terminfo
- Using & Managing PPP
- Using & Managing UUCP
- Virtual Private Networks
- Volume II: X Window System
  - Administrator’s Guide
- Web Security & Commerce
- Windows/NT Backup & Restore
- Windows/NT Desktop Reference
- Windows/NT in a Nutshell
- Windows/NT Server 4.0
  - for Netware Administrators
- Windows/NT SNMP
- Windows/NT User Administration

**Web Review Studio Series**
- Designing Sound for the Web
- Designing with Animation
- Designing with JavaScript
- GIF Animation Studio
- Photoshop for the Web
- Shockwave Studio
- Web Navigation:
  - Designing the User Experience

**UNIX**
- Exploring Expect
- Learning VBScript
- Learning GNU Emacs,
  - 2nd Edition
- Learning the bash Shell,
  - 2nd Edition
- Learning the Korn Shell
- Learning the UNIX Operating System,
  - 4th Edition
- Learning the vi Editor,
  - 5th Edition
- Linux Device Drivers
- Linux in a Nutshell
- Linux Multimedia Guide
- Running Linux,
  - 2nd Edition
- SCO UNIX in a Nutshell
- sed & awk,
  - 2nd Edition
- Tcl/Tk Tools
- UNIX in a Nutshell,
  - Deluxe Edition
- UNIX in a Nutshell,
  - System V Edition
- UNIX Power Tools,
  - 2nd Edition
- Using csh & tcsch
- What You Need To Know:
  - When You Can’t Find Your
  - UNIX System Administrator
- Writing GNU Emacs Extensions

**Windows**
- Access Database Design
- and Programming
- Developing Windows Error Messages
- Excel97 Annoyances
- Inside the Windows 95 File System
- Inside the Windows 95 Registry
- Office97 Annoyances
- VB/VBA in a Nutshell:
  - The Languages
- Win32 Multithreaded Programming
- Windows95 in a Nutshell
- Windows97 Annoyances
- Windows NT File System Internals
- Windows NT in a Nutshell
- Word97 Annoyances

**Using the Internet**
- AOL in a Nutshell
- Bandits on the Information Superhighway
- Internet in a Nutshell
- Smiles
- The Whole Internet for Windows95
- The Whole Internet:
  - The Next Generation
- The Whole Internet
- User’s Guide & Catalog

**Programming**
- Advanced Oracle PL/SQL
  - Programming with Packages
- Applying RCS and SGCS
- BE Developer’s Guide
- BE Advanced Topics
- C++: The Core Language
- Checking C Programs with lint
- Encyclopedia of Graphics File Formats,
  - 2nd Edition
- Guide to Writing DCE Applications
- lex & yacc,
  - 2nd Edition
- Managing Projects with make
- Mastering Oracle Power Objects
- Oracle8 Design Tips
- Oracle Built-in Packages
- Oracle Design
- Oracle Performance Tuning,
  - 2nd Edition
- Oracle PL/SQL Programming,
  - 2nd Edition
- Oracle Scripts
- Porting UNIX Software
- POSIX Programmer’s Guide
- POSIX 4: Programming
  - for the Real World
- Power Programming with RPC
- Practical C Programming,
  - 3rd Edition
- Practical C++ Programming
- Programming Python
- Programming with curses
- Programming with GNU Software
- Putfiles Programming
- Software Portability with iname,
  - 2nd Edition
- Understanding DCE
- UNIX Systems Programming for SVR4

**X Programming**
- Vol. 5M: X Window System User’s
  - Guide, Motif Edition
- Vol. 4M: X Toolkit Intrinsics
  - Programming Manual,
  - Motif Edition
- Vol. 5: X Toolkit Intrinsics Reference
  - Manual
- Vol. 6A: Motif Programming Manual
- Vol. 6B: Motif Reference Manual
- Vol. 8: X Window System
  - Administrator’s Guide

**Software**
- Building Your Own Website™
- Building Your Own Web Conference
- WebBoard™ 3.0
- WebSite Professional™ 2.0
- PolyForm™
UK, EUROPE, MIDDLE EAST AND NORTHERN AFRICA (EXCEPT FRANCE, GERMANY, SWITZERLAND, & AUSTRIA)

INQUIRIES
International Thomson Publishing Europe
Berkeley House
168-173 High Holborn
London WC1V 7AA
United Kingdom
Telephone: +44-171-97-1422
Fax: +44-171-97-1426
Email: itipint@itps.co.uk

ORDERS
International Thomson Publishing Services, Ltd.
Chertion House, North Way
Andover, Hampshire SP10 5BE
United Kingdom
Telephone: +44-264-342-806 (outside UK)
Fax: +44-264-342-832

FRENCH LANGUAGE BOOKS
All countries except Canada
Telephone: +33-01-44-41-11-44
Fax: +33-01-44-41-11-46
Email: godif@eyrolles.com

GERMANY, SWITZERLAND, AND AUSTRIA

INQUIRIES
O'Reilly Verlag
Balthasarstr. 81
D-50670 Köln
Germany
Telephone: 49-221-97-31-60-0
Fax: 49-221-97-31-60-8
Email: anfragen@oreilly.de

ORDERS
International Thomson Publishing
Königswinterer Straße 418
53227 Bonn, Germany
Telephone: 49-228-97024 0
Fax: 49-228-41342
Email: order@oreilly.de

JAPAN
O'Reilly Japan, Inc.
Kiyoshige Building 2F
12-Banchi, Sanei-cho
Shinjuku-ku
Tokyo 160-0008 Japan
Telephone: 81-3-3356-5227
Fax: 81-3-3356-5261
Email: kenji@oreilly.com

INDIA
Computer Bookshop (India) Pvt. Ltd.
190 Dr. D.N. Road, Fort
Bombay 400 001 India
Telephone: 91-22-207-0989
Fax: 91-22-262-3551
Email: cbsbom@giashm01.vsnl.net.in

HONG KONG
City Discount Subscription Service Ltd.
Unit D, 3rd Floor, Yan's Tower
27 Wong Chuk Hang Road
Aberdeen, Hong Kong
Telephone: 852-2580-3539
Fax: 852-2580-6463
Email: citydis@ppn.com.hk

KOREA
Hanbit Media, Inc.
Yeksam-dong, Bldg. 202
97-62, Kangnam-dong
Seoul, Korea
 Telephone: 822-556-0363
Email: hant93@chollian.dacom.co.kr

SINGAPORE, MALAYSIA, AND THAILAND
Addison Wesley Longman Singapore PTE Ltd.
25 First Lok Yang Road
Singapore 629734
Telephone: 65-268-2666
Fax: 65-268-7023
Email: daniel@longman.com.sg

PHILIPPINES
Mutual Books, Inc.
429-D Shaw Boulevard
Mandaluyong City, Metro Manila, Philippines
Telephone: 632-725-7538
Fax: 632-721-3056
Email: mbilikog@mnl.sequel.net

CHINA
Ron's DataCom Co., Ltd.
79 Dongwu Avenue
Dongshiku District
Wuhan 430040
China
Telephone: 86-27-3892568
Fax: 86-27-5222108
Email: hongfeng@public.wh.hb.cn

ALL OTHER ASIAN COUNTRIES
O'Reilly & Associates, Inc.
101 Morris Street
Sebastopol, CA 95472 USA
Telephone: 707-829-0515
Fax: 707-829-0104
Email: order@oreilly.com

AUSTRALIA
Woodslane Pty. Ltd.
7/5 Vuko Place, Warriewood NSW 2102
P.O. Box 935
Mona Vale NSW 2103
Australia
Telephone: 61-2-9970-5111
Fax: 61-2-9970-5002
Email: info@woodslane.com.au

NEW ZEALAND
Woodslane New Zealand Ltd.
21 Cooks Street (P.O. Box 575)
Wagantui, New Zealand
Telephone: 64-6-347-6543
Fax: 64-6-345-4840
Email: info@woodslane.com.au

THE AMERICAS
McGraw-Hill Interamericana Editores,
S.A. de C.V.
Cedro No. 512
Col. Adampar 06450
Mexico, D.F.
Telephone: 52-5-541-3155
Fax: 52-5-541-4913
Email: mcgraw-hill@infosel.net.mx

SOUTH AFRICA
International Thomson Publishing
South Africa
Building 18, Constantia Park
138 Sixteenth Road
P.O. Box 2459
Halfway House, 1685 South Africa
Telephone: 27-11-805-4819
Fax: 27-11-805-3648

O'REILLY™

TO ORDER: 800-998-9938 • order@oreilly.com • http://www.oreilly.com/
Our products are available at a bookstore or software store near you.
FOR INFORMATION: 800-998-9938 • 707-829-0515 • info@oreilly.com
Nineteenth century wood engraving of a bear from the O'Reilly & Associates Nutshell Handbook® Using & Managing UUCP.
Cracking DES
Secrets of Encryption Research, Wiretap Politics & Chip Design

Sometimes you have to do good engineering to straighten out twisted politics. The Electronic Frontier Foundation has done so by exploding the government-supported myth that the Data Encryption Standard (DES) has real security.

National Security Agency and FBI officials say our civil liberties must be curtailed because the government can’t crack the security of DES to wiretap bad guys. But somehow a tiny nonprofit has designed and built a $200,000 machine that cracks DES in a week. Who’s lying, and why?

For the first time, the book reveals full technical details on how researchers and data-recovery engineers can build a working DES Cracker. It includes design specifications and board schematics, as well as full source code for the custom chip, a chip simulator, and the software that drives the system. The US government makes it illegal to publish these details on the Web, but they’re printed here in a form that’s easy to read and understand, legal to publish, and convenient for scanning into your computer.

The Data Encryption Standard withstood the test of time for twenty years. This book shows exactly how it was brought down. Every cryptographer, security designer, and student of cryptography policy should read this book to understand how the world changed as it fell.

"Beautifully milks many sacred cows of their crypto policy. EFF exposes more of the emperor’s new clothes, reaching new levels of truth."

—Peter Neumann, Moderator of the Risks Forum

“A very impressive piece of work. This book will change the history of cryptography.”

—Steve Bellovin, co-author of Firewalls and Internet Security

“If the government was honest, the crypto debate would be over by now. EFF’s research conclusively refutes their propaganda.”

—Bruce Schneier, President of Counterpane Systems