Technical Report No. 496

Arriving at FPGA based Hardware Unix-Encription using Iterated Codesign Methods

by

Ingo Cyliax
Steven D. Johnson
Bhaskar Bose, Derivation Systems, Inc., Carlsbad CA

October 1997

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

Arriving at FPGA Based Hardware Unix-Encryption Using Iterated Codesign Methods

Ingo Cyliax, Computer Science, Indiana University
Steven D. Johnson, Computer Science, Indiana University
Bhaskar Bose, Derivation Systems Inc., Carlsbad CA

ABSTRACT: SRAM based FPGAs are well suited for using iterative hardware/software codesign techniques to derive hardware implementations from software algorithms. In this paper we present a case study of the Unix password encryption algorithm implemented in a FPGA using this technique. We have found that: 1. FPGAs are cost effective for accelerating custom algorithms such as Unix *crypt*, 2. SRAM based FPGA are suitable for secure implementations of hardware, and 3. software algorithms can be implemented swiftly in FPGAs using iterative codesign techniques.

KEYWORDS: encryption, DES, FPGA, security, UNIX

1 Introduction

Recently much attention has been given to FPGA based reconfigurable computing. This paper describes a case study of how we implemented the Unix password encryption algorithm (*Unix-crypt*) in a FPGA based coprocessor using iterative hardware software codesign techniques.

We have discovered that Unix-crypt is a very cost effective application for FPGAs which may have a real life application in Unix system and network system administration.

We also believe that using SRAM based FPGA technology makes for a more secure implementation of sensitive applications, such as encryption technology.

Even though this project started as a design example for using VHDL in the classroom, we discovered that several of the entry level VHDL tools we had access to, were not able to synthesize this algorithm when presented as a high level model. We show that using a iterative operator driven hardware/software codesign methodology, staging, can be effective when implementing complex software algorithms in hardware designs.

2 Background

Unix password encryption (Unix-crypt) is implemented as a hash function based on the DES algorithm. The user's password is used as a key to encrypt a block of zeroes into an encrypted string which is stored in a system file. Whenever the user needs to authenticate using his/her password, this process is repeated and the encrypted zero block is compared to the stored value.

The standard DES algorithm is modified by iterating DES 25 times, making brute-force software attacks difficult unless a large number of computers are used to attack a single Unix password. The function is also *salted* in one of 4096 ways to prevent the use of commercial off the shelf (COTS) DES chips to implement hardware password crackers and make it expensive to precompute plaintext to encoded text dictionaries [8].

A common attack on Unix systems is to obtain the system's password file and use a plaintext dictionary key search to attack passwords. Once weak password have been discovered, the attacker can use this knowledge to launch an attack on more secure systems on which users may have accounts [6]. Using a dictionary password key search attack is one of the methods which the Internet Worm of 1988 used to gain access to other systems [10].

The standard Unix-crypt function available in the Unix standard C library (*lib-c*) on domestically distributed Unix systems. Source code for the Unix-crypt function (in C language) can be found in the source distributions of Linux, FreeBSD and NetBSD. Also, an optimized version of Unix-crypt (fcrypt) [5] is available in several security packages such as COPS [4] and Crack [9].

A responsible Unix system and network administrator uses dictionary attacks to scan stored password files for weak passwords. These scans are run periodically, but can take a long time when a large number of passwords are involved [4, 9]. For example, 32000 accounts, a typical number of accounts at a medium sized university, may take 40 hours to check against 500,000 plain texts on a 200Mhz Pentium-Pro based system.

A dictionary of 500,000 entries is small, since it would typically include variants of the same string with difference case encodings. Typical scanning runs may use dictionaries of up to 1 Million plaintext words, which are tried with different capitalizations, and sometimes substitutions for expressions like "to","too" and "2", and combinations of smaller words. Large dictionary scans may take months of compute time [6]. Administrators are reluctant to run password scanning on high end research computing facilities, whose

facilities are typically charged for the CPU time they, themselves, consume.

In a nutshell, it takes valuable resources for system administrators to run password scans. An attacker can get CPU cycles for free on systems which have already been compromised.

Attempts have been made to build hardware implementations of the Unix-crypt function in order to make password scanning more cost- effective for system administrators. One such implementation by Leong and Tham is a ECL based board level design. This machine cost about $2000 to build and was able to run 166Kcrypt/sec.

This is compared to a software implementation running on an RS/6000, which runs at 830crypt/sec but costs $10,000—a typical high end workstation that a system administrator would have had access to in 1991 [7].

In this paper we will use the metric

$$cps\$ = \frac{crypts}{seconds \times dollars}$$

to make cost-performance comparisons of various Unix-crypt implementations. The RS/6000 mentioned above measures 0.083cps$, compared to the ECL implementation at 83cps$. Today, a 200Mhz Pentium-Pro based system costing $3000 runs at 14Kcrypt/sec, or 4.6cps$.

3 Implementation

Unix-crypt is based on a modified DES algorithm.

The DES algorithm uses 16 iterations (*rounds*) to encode a 64-bit input block into a 64-bit output block. A 56-bit input key is used to calculate 16 48-bit subkeys, one for each round, using the key schedule which we describe later. Figure 1 shows general overview of the DES algorithm.

The 64-bit input block is first permuted using the IP permutation. A permutation changes the bit ordering of the input string and is specified as a table enumerating the input position for a particular output bit. Table 1 shows the bit encoding for the IP permutation. Similarly, the output block is divided into two 32-bit halves, which are swapped before being permuted with the IP^{-1} permutation after all 16 rounds have been completed.

Each round computes on a 32-bit subblock. An E permutation expands the 32-bit subblock into a 48-bit string by changing the bit ordering and duplicating some of the bits. At this point, the 48-bit subkey for the current round is XOR'ed with the expanded subblock. The resulting 48-bit product

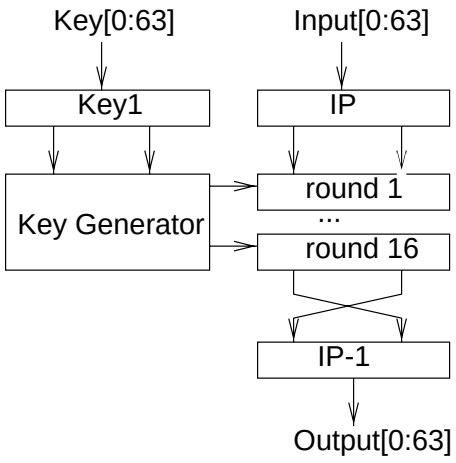

Figure 1: Structure of the DES algorithm

	–0	*–1*	*–2*	*–3*	*–4*	*–5*	*–6*	*–7*
0–	58	50	42	34	26	18	10	2
1–	60	52	44	36	28	20	12	4
2–	62	54	46	38	30	22	14	6
3–	64	56	48	40	32	24	16	8
4–	57	49	41	33	25	17	9	1
5–	59	51	43	35	27	19	11	3
6–	61	53	45	37	29	21	13	5
7–	63	55	47	39	31	23	15	7

Table 1: The initial permutation (IP) table.

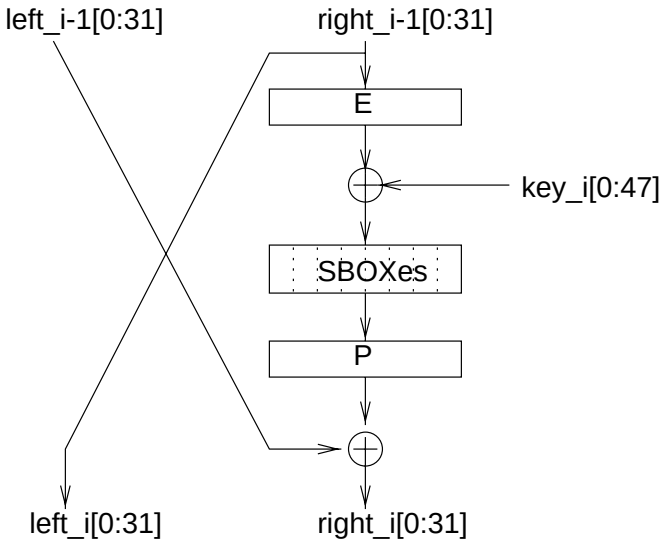

Figure 2: A single round of DES

is grouped into 8 6bit subgroups. Substitution boxes then reduce the 8 groups of 6bits into 8 groups of 4bits. A substitution box (SBOX) is a look up table which output a code for each input code. The results from the SBOXes are then merged back into a 32-bit string. This string is permuted with the P permutation before being XOR'ed with the other half of the intermediate result from the last round. Figure 2 shows the details of a single round.

A 64-bit input key is reduced to a 56-bit key by stripping 8 parity bits and permuting the remaining 56-bit using the Key1 permutation. This 56key is split into 2 28-bit halves each of which can be rotated by either 1 or 2 positions for each round. The sequence of rotations, or *key-schedule*, is specified as a string $\{1, 1, 2, \ldots, 2, 1\}$, where each member specifies the number of rotations for that round. It should be noted that the sum of all rotations is 28, thus the state of the 2 subkeys reverts to the initial state after 16 rounds. On each round, the 2 28-bit key halves are merged into a 56-bit string and reduced to a 48-bit subkey using the *key2* permutation [11]. Figure 3 shows the key calculation details.

Several chip level implementation of DES exist commercially. To prevent the use of COTS DES chips, the Unix-crypt algorithm introduces a salted permutation for the expanded input in each round. The salt, a 12 bit number,

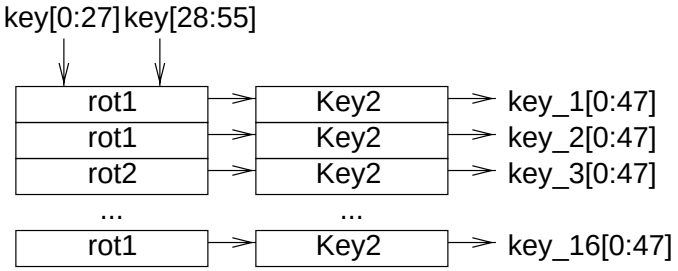

Figure 3: Key calculation

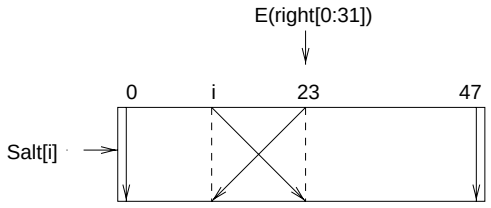

Figure 4: The salt box

permutes 24 bits of the 48-bit string by either swapping a pair of bits or not depending on the value of the corresponding salt bit. Figure 4 shows the salt box.

In addition to the salted permutation, the Unix-crypt algorithm iterates the standard DES algorithm 25 times, i.e. 400 rounds total [8, 7].

Our initial implementation of the Unix-crypt, based on the DES algorithm described by Tanenbaum [11], was done in VHDL using high level RTL form. Several VHDL implementations of various Unix-crypt configurations (sequential vs. pipelined) were successfully simulated against test vectors derived by running a program based on the C library version of Unix-crypt which calculated encrypted strings based on random keys.

Synthesizing a FPGA design from the VHDL did not work because our VHDL tools were not capable of synthesizing such a complex design. This was due to memory limitations, since some of the the expressions are very large.

Since we were not able to synthesize an FPGA implementation from our

VHDL model, we used an alternate method. We iteratively implemented subcomponents of the design in hardware and used software components to exercise the hardware components. These hardware and software codesigns implement the complete algorithm which are tested against the C library version by comparing vectors.

We first implemented the essential permutation and substitution elements in the FPGA. Permutations were generated using a Unix *awk* script which read the permutation tables and generated a netlist connecting input signal `i[0-n]` to output signals `o[0-m]` using Xilinx buf gates. The substitution lookup tables were implemented as Xilinx ROM cells using the *memgen* utility supplied with the XactStep software. All the data movement and control was done with software.

Once the individual functional units were tested, we implemented the complete datapath of the chip. The data path consists of the a 64-bit X register which is used to hold the input, output and intermediate block, and a 56-bit K register, which holds the key. The datapath for the X register can compute IP, IP-1 or one round of the modified Unix crypt, f_crypt(). The K register data path can compute either a 1 bit rotation or a 2 bit rotation. There is also a S register which holds the salt value. Finally, a control register selects the operation codes for the X and the K data path. Software was now only used to load the registers, and then to control the data path by loading the control register with appropriate function codes. Figure 5 shows the block diagram the hardware implementation, and Figure 6 shows the control software component.

Finally, the control was realized in hardware as a state machine. A PLD assembler was used to synthesize the state machine for the control. The source level description for the assembler is essentially a one-to-one translation of the C code. The control register now implements a start command while the status register allows the CPU to monitor the status of the hardware, including when an encryption process has finished. The software to drive this completed hardware implementation can be seen in Figure 7.

The Unix C library *crypt* function was used to check the hardware and software codesign implementation and finally the hardware-only implementation by comparing vectors. The test vectors were calculated on random keys using the Unix *lib-c crypt(3)* function. Also, some keys found to exhibit pathological problems in the VHDL simulation were included.

The design was implemented on a Xilinx XC4010PC191-100. We used this chip to compare the timing estimates from the XactStep software to

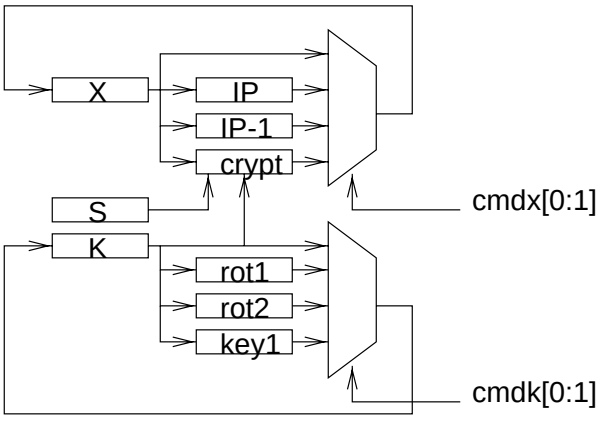

Figure 5: Hardware architecture

actual measurements. Once we were satisfied the timing specification was accurate, we generated configurations and extracted timing information for various XC4010 and XC4008 chips.

The XC4010PC191-10 was interfaced to an Intel 486DX2 based workstation running BSDi, a commercial BSD Unix implementation, through it's parallel port. The protocol used for this design is based on the Logic Engine protocol [13] which we use in our instructional and research prototyping system. The appliction interface enables a C (or Scheme) program to read and write the X, K, and S registers as well as the control and status register in the chip. Figure 8 shows the experiment.

4 Results

We have implemented the Unix-crypt algorithm in a Xilinx XC4010PG191-10 chip using hardware software staging. We have also tested its functionality using test vectors which were generated using a reference software implementation.

The XactStep timing extractor predicted this chip to execute at 120ns ffs–ffs speed. This represents the worst path through the datapath. We have tested this chip at up to 16Mhz before it started failing the software generated test vectors. We are confident that the XactStep timing estimates are conservative and that the chip will run at the estimated speed (8.33Mhz)

```
do_hwcrypt(fd,salt,key,x)
  int fd;
  char *key;
  char *salt;
  char *x;
{
  int j;

  /* start off with a clean slate, 0 -> x  */
  crypt_reset(fd);

  /* load registers */
  crypt_set_k(fd,key);
  crypt_set_s(fd,salt);
  crypt_step(fd,CMD_KEY1,CMD_HOLDX);

  /* run machine */
  for(j=0;j<25;j++){
    crypt_step(fd,CMD_ROT1,CMD_IP);
    crypt_step(fd,CMD_ROT1,CMD_FUNCX); crypt_step(fd,CMD_ROT2,CMD_FUNCX);
    crypt_step(fd,CMD_ROT2,CMD_FUNCX); crypt_step(fd,CMD_ROT2,CMD_FUNCX);
    crypt_step(fd,CMD_ROT2,CMD_FUNCX); crypt_step(fd,CMD_ROT2,CMD_FUNCX);
    crypt_step(fd,CMD_ROT2,CMD_FUNCX); crypt_step(fd,CMD_ROT1,CMD_FUNCX);
    crypt_step(fd,CMD_ROT2,CMD_FUNCX); crypt_step(fd,CMD_ROT2,CMD_FUNCX);
    crypt_step(fd,CMD_ROT2,CMD_FUNCX); crypt_step(fd,CMD_ROT2,CMD_FUNCX);
    crypt_step(fd,CMD_ROT2,CMD_FUNCX); crypt_step(fd,CMD_ROT2,CMD_FUNCX);
    crypt_step(fd,CMD_ROT1,CMD_FUNCX); crypt_step(fd,CMD_HOLDK,CMD_FUNCX);
    crypt_step(fd,CMD_HOLDK,CMD_IPINV);
  }
    /* read result */
    crypt_read(fd,x);
}
```

Figure 6: The software component to control the FPGA datapath.

```
do_hwcrypt(fd,salt,key,x)
   int fd;
   char *key;
   char *salt;
   char *x;
{
   /* reset machine, this clears the X register.
   crypt_reset(fd);
   /* load the key and salt register */
   crypt_set_s(fd,salt);
   crypt_set_k(fd,key);
   /* start machine and wait for completion */
   crypt_go(fd);

   while(crypt_done(fd))
     ;

   /* read answer */
   crypt_read(fd,x);
}
```

Figure 7: Software to drive FPGA based Unix-Crypt coprocessor.

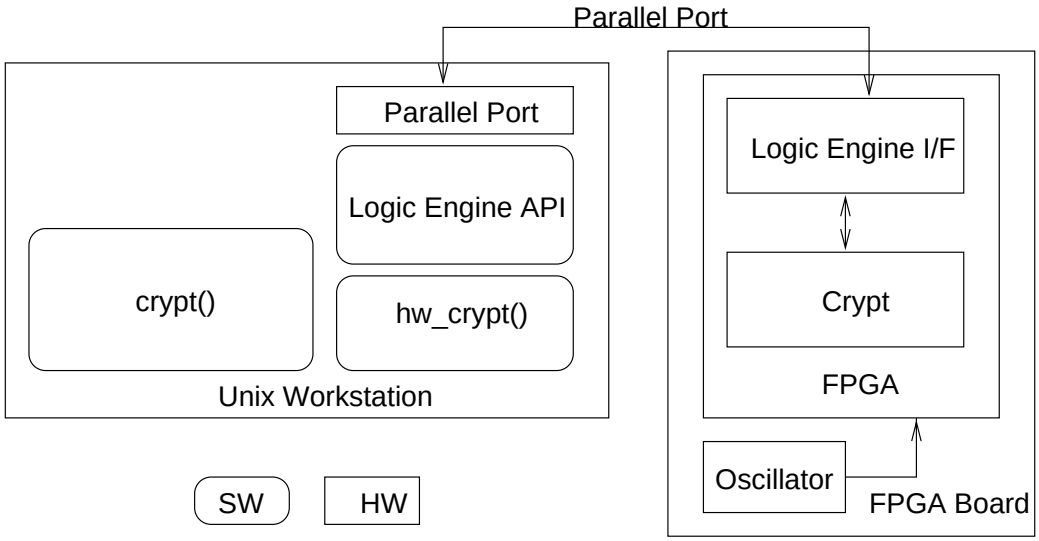

Figure 8: The test environment

PART	COST(9/97)	FFS–FFS	CPS	CPS$
XC4010-6PG191	334	69ns	34K	101
XC4010PG191-10	200	120ns	20K	100
XC4010E-2PC84	109	36ns*	69K	633*
XC4010E-3PC84	81	41ns	60K	740
XC4010E-4PC84	62	48ns	52K	838
XC4010XL-3PC84	55	41ns*	69K	1254*

*extrapolated estimates

Table 2: FPGA cost-performance comparison

CPU	COST	CPU×SPEED	CPS$
PentiumPro200	3000	14000	4.60
SGI Origin2000	277000	8×11820	0.34
Sparc Ultra200	12000	2×10500	1.75
Sparc Ultra170	7500	2× 7621	2.00

Table 3: CPU cost-performance comparison

over the complete temperature and Vcc range.

We have extracted timing estimates for a variety of Xilinx chips of different speed ratings for the XC4000E technology. Table 2 shows a comparison of the various chips compared to their cost.

We have run reference software implementation [12] on a variety of CPUs and list their performance and cost in Table 3.

While the design provides adequate performance, a significant speedup can be achieved when the salt function is implemented statically. Implementing the salt function statically, eliminates a 2:1 MUX in the X datapath, which speeds the design up by approximately 10 per cent. Furthermore, the design is less complex and can be implemented in a smaller chip. Figure 4 summarizes the cost/performance improvement of implementing the salt statically.

Using the Logic Engine API over the PC parallel port to communicate with the chip, we have attained a sustainable transfer rate at 7Kcrypt/sec. The we chip used runs at 20Kcrypt/sec peak; this indicates only a 20 per cent utilization of the encryption chip in our experiment.

PART	COST (9/97)	FFS–FFS	CPS	CPS$
XC4010-6PG191	334	61ns	40K	122
XC4010E-3PC84	81	39ns	64K	791
XC4010E-4PC84	62	41ns	60K	983
XC4008E-3PC84	69	38ns	65K	953
XC4008E-4PC84	53	41ns	60K	1132

Table 4: Cost performance estimates for static salt.

5 Conclusions

We found that Xilinx XactStep timing estimates are conservative and very consistent. We were able to duplicate the timing estimates in several place&route runs, each of which was initialized with a different seed. Overall, we found the XactStep core package to be very robust running under SunOS on Sun Sparcs.

FPGAs are well suited for cost effectively implementing custom software algorithms. In this case, the FPGA implementation drastically outperforms CPU implementations. The most cost effective software implementation tested is 4.6cps$, while the most cost effective Xilinx FPGA implementation we have evaluated was 1254cps$.

In our performance comparisons, we compare a CPU based system to a FPGA chip. We feel that this is a fair comparison. A high performance CPU chip, such as as Pentium, needs support hardware such as cache memory and memory in order to function. A FPGA implementation, however, could run independently with little support hardware, e.g. if doing a brute force key search.

While library driven high level VHDL synthesis works for standard designs, such as DSP applications and CPU datapaths, etc., it still has some trouble for large random logic designs, such as encryption or encoding chips. For these design, a netlist style VHDL implementation would be easier to synthesize.

We have discovered that for this design example, using a staging approach to do hardware synthesis resulted in an implementation which was correct and had acceptable performance. The time was spent actually synthesizing and testing the design components, rather than tweaking the model to make an automatic synthesizer generate a design which passed.

We have been working on tools which can formalize an operator driven iterative synthesis by verifying each derivation. These tools have been used to synthesize object as complex as a 32-bit RISC CPU [2, 3]. Derivation Systems Inc, is currently working on derivations of this chip as well as PCI cores using their formal derivation system, DRS.

For this application, the SRAM based FPGA may represent a more secure implementation than a hard-wire implementation or the software implementation. In this scheme, the configuration for the FPGA is stored in encrypted form on the hard disk of the machine which interfaces to the encryption hardware. An operator would have to be present in order to provide a key/password to configure the FPGA. Once the FPGA has been configured and readback has been disabled, it would be difficult to reverse engineer the design from the FPGA, or obtain the crypt machine for illicit purposes with the design still intact. I.e. once the power to the FPGA has been lost, it needs to be reconfigured using the key/password which protects the configuration file.

SRAM based FPGAs are suitable for optimizing solutions by parameterizing designs into several simpler designs and dynamically reconfiguring the hardware. In this application, a more optimized version of this machine with static salt functions can be implemented and stored on disk. When the machine for a specific salt is needed, it is simply configured. It is possible to prebuild all 4096 implementations of the Unix-crypt, one for each salt, and store them on a CDROM.

The PC parallel port protocol we currently use is not optimized for Unix-crypt type applications. A Unix-crypt specific protocol may be able to reach transfer speeds at up to 37Kcrypt/sec. Even at this speed, the parallel port would present bottle neck for hardware based Unix-crypt implementation. We are investigating a PCI or ISAbus implementation of this design.

The implementation described in this paper is very straightforward and naive. We would like to investigate how techniques such as floor planning and new parallel bit serial algorithms as presented in [1] effect the cost of Unix-crypt and other password encryption or hash algorithms on FPGA implementations. This will also present us with more case studies to use in our research and classroom.

Finally, since it is reasonable to assume that if the good guys (us) are looking at FPGA implementations of Unix-crypt for password scanning purposes, the bad guys are probably also looking at it for cracking purposes. A $3000 investment in FPGA hardware, equivalent to a high end Pentium

system, would allow someone to launch a brute force key search attack on a single password of 6 alphanumeric characters (62^6) in approx 4.0 hrs.... Make sure you choose those strong passwords that your system administrators have been bugging you about!

References

[1] Eli Biham. A fast new des implementation in software. Technical report, Israel Institute of Technology, Haifa, Isreal, 1996.

[2] Bhaskar Bose. DDD/FM9001 – derivation of a verified microprocessor. Technical Report 456, Indiana University Computer Science Department, Bloomington, Indiana, April 1996. PhD. dissertation, accessable via `www.cs.indiana.edu`.

[3] Bhaskar Bose and Steven D. Johnson. DDD-FM9001 – derivation of a verified microprocessor. Technical Report 380, Indiana University Computer Science Department, Bloomington, Indiana, April 1996.

[4] Dan Farmer. Cops version 1.04+ `readme` file, 1991.

[5] Michael Glad. UFC-Crypt `readme` file, 1992.

[6] Daniel V. Klein. 'foiling the cracker': A survey of, and improvements to, password security. Technical report, Software Engineering Institute, Carnege Mellon University, 1990.

[7] Philip Leong and Chris Tham. Unix password encryption considered insecure. In *USENIX*, Winter 1991.

[8] Robert T. Morris and Ken Thompson. Password security: A case history. *Communications of the ACM*, 22(11):594–597, November 1979.

[9] Alec D.E. Muffett. A sensible password checker for unix. Technical report, Computer Unit, University College of Wales, 1992.

[10] Eugene H. Spafford. The Internet Worm program: An analysis. Technical Report CSD-TR-823, Purdue University, November 1988.

[11] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1981.

[12] Tom Truscott, 1995. *crypt(3)* routine from NetBSD 1.1 distribution, based on algorithms from Robert W. Baldwin and James Gillogly.

[13] Robert M. Wehrmeister. Logic Engine user's manual. Laboratory manual, Indiana University Computer Science Department, 1991. `http://www.cs.indiana.edu/classes/p442/man/le_manual`.