# Web Server Protection by Customized Instruction Set Encoding

Bernhard Fechner, Jörg Keller, and Andreas Wohlfeld

FernUniversität in Hagen
FB Informatik — LG Parallelität und VLSI
58084 Hagen, Germany
{bernhard.fechner,joerg.keller,andreas.wohlfeld}@fernuni-hagen.de

## Abstract

*We present a novel technique to secure the execution of a processor against the execution of malicious code (trojans, viruses). The main idea is to permute parts of the opcode values so that it gets a different semantic meaning. A virus which does not know the permutation is not able to execute and will cause a failure such as segmentation violation, whereby the execution of malicious code is prevented. The permutation is realized by a lookup table. We develop several variants that require only small changes to microprocessors. We sketch how to bootstrap a system such that all intended applications (including operating system) are reversely permuted, and can execute as intended. While this will be cumbersome for typical personal computers, it will work for web servers, because the number of applications and frequency of installation is lower. Furthermore, web servers are particularly endangered: they cannot be protected as good as personal computers, because by the very nature of their duty they are more openly connected with the internet than any other computer in an organization's network.*

## 1. Introduction

Today's computer systems face a huge amount of threats (malicious e-mail, viruses, worms and trojan horses) causing huge costs. For a classification of computer viruses, see [5]. Table 1 shows the estimated costs caused by worms [3]. The huge costs result from a rapid infection rate [9, 10].

| Program | Est. cost [$10^9$ US$] |
|---|---:|
| Melissa | 1.10 |
| ILOVEYOU | 8.75 |
| CodeRed | 2.62 |
| SirCam | 1.15 |
| Nimda | 0.63 |

**Table 1. Estimated Cost per Worm.**

Web servers are particularly in danger, as they are on the forefront of any organization's network. Typically, they are only separated by one firewall from the internet, being placed in the so-called DMZ (demilitarized zone). In 2001, the worm "Code-Red v2" probed randomly chosen Internet hosts for a vulnerability in the Microsoft Internet Information Service (IIS) web server. Infected hosts tried to contaminate other hosts reaching an infection rate of almost 360,000 hosts in no more than 14 hours (2,000 hosts per minute before peaking [9]). The direct costs of recovering from Code-Red and its derivations have been estimated to US$ 2.6 billion [3]. To fight this harassment seems to be an everlasting battle in which the developers of malicious software are always one step ahead. Contemporary computer systems have at least a personal firewall, a virus scanner or an intrusion detection system installed. The tools have been developed since the late eighties [2, 4]. However, a firewall is of no use if a valid protocol (typically http via port 80) is used and an application vulnerability is exploited. A virus scanner often is useless as well, as it can only detect a virus whose pattern is present in its data base, which is updated at most once a day from a central database, where the pattern can only be added one or a few days after the occurrence of the virus (so-called zero-day response problem). If the virus performs only legitimate actions, such as reading the current user's address book, and sending emails, the intrusion detection system also cannot help.

The aim of this work is the development of a code permutation which enables the defenders of secure computing systems to prevent malicious code from executing and detect where that code is located. Our main idea is to provide a web server with a unique instruction set encoding, and appropriately compiled applications, such that a virus that does not know the particular encoding has no chance to execute on that machine. Execution of a virus code will almost immediately violate some operating system protection means, which stops the execution and creates a log file entry. We are aware that this measure does not help against a macro virus, but it is a first step. We are also aware that application of our idea may be too cumbersome for normal personal computers. Web servers are special because the installed code base is much smaller than in an ordinary desktop PC, and the rate of change much slower.

We will reveal that the changes in a microprocessor to accommodate a personalized instruction set encoding are small. We will detail how to provide appropriately encoded applications, and we will report on the results of some preliminary experiments.

The idea to personalize instruction set encodings is not completely new. In [1] a binary-to-binary translator is used at load time of an application to scramble the opcodes, to prevent binary code injection attacks. As this approach considers any executable on the local disk as trustworthy, it cannot protect against viruses. In [8] a technique to change the semantics of opcodes according to the state of a finite state machine running as part of the application is used. Yet, this can only be used to protect against code re-engineering, and is not directly applicable to prevent codes from executing. In [12] and the many references it contains, measures to protect a hardware from physical tampering are revealed. We assume that the physical integrity of the web server is guaranteed by organizational measures.

The remainder of this work is organized as follows. Section 2 presents how personalized instruction set encoding can be efficiently implemented in microprocessors. In Section 3 we develop some schemes how such a modified hardware could be used with a standard operating system. We present some preliminary experimental results in Section 4. Section 5 concludes the paper.

## 2. Permuted instruction set encoding

An executable in contemporary computer systems consists of a sequence of assembler instructions encoded as opcodes. The opcodes are typically standardized in order to allow executables to run unchanged on as many platforms as possible, the perhaps most prominent example being the IA32. If on the other hand a processor encodes the assembler instructions in a unique way, then executables can only be executed with the knowledge of this encoding. Executables compiled for the standard encoding will be misinterpreted and will crash. While there are many ways to change an instruction set, we would like to keep the format of assembler instructions as they are and only modify the opcode encoding. This also allows us to keep all software tools down to assembler level unaltered.

Consider as an example a very simple microprocessor with only four types of instructions: Load (00), Store (01), Compute (10), Jump (11). We now apply a permutation $\pi : \{0,1\}^n \rightarrow \{0,1\}^n$ on the encodings, $n = 2$ being the number of bits of an opcode, such as $\pi(x) = x + 1 \bmod 4$. If we change the processor hardware accordingly, we get an encoding Load (01), Store (10), Compute (11), Jump (00). An executable compiled for the original encoding will have to be subjected to a binary-to-binary translator that performs the inverse permutation $\pi^{-1}$ on the opcodes. Then the modified executable will execute on the modified microprocessor as the unmodified executable would execute on the unmodified microprocessor. An unmodified executable executed on the modified microprocessor would form a weird code sequence and very soon violate some operating system protection means. The permutation forms a kind of secret key, and only properly encrypted executables will be properly decrypted during execution.

The easiest way to incorporate the opcode permutation into a microprocessor is to provide a lookup table of modified opcodes, indexed by the unmodified opcodes. In this manner, only the decoder has to be modified. Moreover, this implementation will be applicable to most microprocessors, with hardwired control, microprogramming, or a hybrid approach. Lookup tables are widely used in hardware cryptography, see e.g. [6]. Unfortunately, an additional table lookup during decoding might either mean another pipeline stage, which would require a major re-design of the processor, or would slow down the cycle time. A simpler scheme can be obtained by restricting the type of permutation. One possibility is to only permute the $n$ bits of the opcode. Thus, the permutation $\pi$ is realized via a second permutation $\rho : \{0,\ldots,n-1\} \rightarrow \{0,\ldots,n-1\}$

by performing

$$\pi(o_{n-1}, \ldots, o_0) = o_{\rho(n-1)}, \ldots, o_{\rho(0)} \ ,$$

where $o_{n-1}, \ldots, o_0$ represents the opcode.

A bit permutation can be realized in hardware by a permutation network such as a Benes-Network [7], which is a circuit with depth $2 \cdot \log(n) - 1$ to permute $n$ bits. For an 8-bit opcode, the depth would be only 5, which is a much lower overhead than a 256-byte lookup table access. Yet, this will lead to the situation that some opcodes remain unchanged, such as opcodes consisting of only zeroes or only ones. An even simpler scheme can be obtained by choosing a non-zero bit string $a = a_{n-1}, \ldots, a_0$ of the same length as the opcode, and performing a bitwise exclusive or between $a$ and each opcode to obtain the permuted opcode:

$$\pi(o_{n-1}, \ldots, o_0) = o_{n-1} \oplus a_{n-1}, \ldots, o_0 \oplus a_0 \ .$$

The overhead here reduces to the depth of an exor gate. These simplified implementations still work for hardwired and microprogrammed microprocessors. Furthermore they increase the chance to integrate the permuting into the existing decode cycle, without a substantial slowdown of the cycle time.

A comparison reveals the following: if opcodes consist of $n$ bits, then the lookup table variant provides $(2^n)!$ different permutations, while the permutation-network variant provides $n!$ permutations, while the exor variant provides $2^n$ different permutations. For $n = 8$, the number of permutations for the first variant is about $2^{2048}$, whereas for the latter two variants it is only $40,320$ and $256$, respectively. While the simple schemes perform well for an ordinary virus, the number of permutations is too small to protect against a virus that is aware of such a measure. Such a virus could try to spread in a large number of copies that use all possible permuted encodings, so that in the end one copy of the virus would succeed. To protect against such viruses, the measure can be complemented by another change, such as exchanging the positions of two register arguments in the code, to increase the number of encodings to a level which makes it unfeasible for a virus to spread in such a number of different encodings. Still, even a small number of possible opcode encodings forces a permutation-aware virus to spread in many different variants with small chances of success, thus slowing down its rate of infection.

A quite different idea is to embed the personalization deeper into the microprocessor. This makes it more efficient but requires more changes. We exemplify this idea for microprogrammed processors: we permute the entry points into the microcode ROM. While executing, the processor fetches instructions as usual from the main memory. Instructions are being decoded into micro-ops. Each microcode entry contains a pointer to the start of the instruction specific microprogram. This microprogram can consist of one or multiple micro-ops. This pointer is looked up in a permutation table. According to the stored permutation, the new microprogram start address will be computed. The permuted start address will select the correct micro-op from the microcode which will be then scheduled for execution. To compute the size of the lookup table, we consider a microcode ROM with entries of length $e$ and $m$ entries. The pointer to a micro-op has therefore the length $\log_2 m$. Each entry in the permutation lookup table contains two pointers: the original start address used for the lookup and the start address according to the permutation. This allows to switch back to unmodified execution, and additionally forms a kind of associate memory, because not every start address will be a valid index to the lookup table. Therefore, the size of the lookup table will be $2 \cdot m \cdot \log_2 m$. We may also integrate the lookup table in the microcode ROM, which will save half of the space and one lookup, because the table only needs to store the permuted start address, and we can use the opcode as an index to both memories (microcode ROM and lookup table), and perform both accesses in parallel. This still allows to switch between permuted and non-permuted opcode encoding, and increases path length only by a multiplexer to select between both start addresses. Figure 1 details the procedure for the general case. Note that loading the permutation either requires knowledge of the original start addresses, or requires an installation circuit that reads out each start address $ad_i$ and stores it at entry $\pi(i)$ of the lookup table.

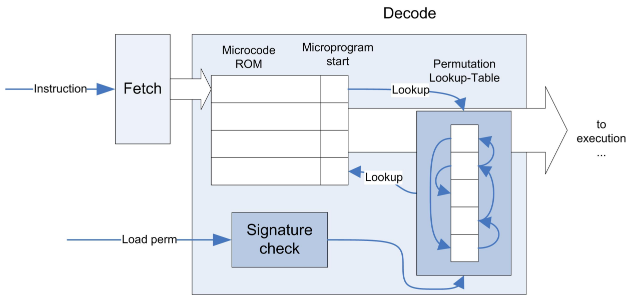

**Figure 1. Enhanced microcode execution.**

## 3. System issues

When deploying a microprocessor with a modified opcode encoding, the question is how to start the system and how to provide appropriate software. At the start of the processor's execution, an initial permutation configuration is activated. Usually this will be a permutation realizing identity. This enables the vendors to keep their power-up boot code unaltered. Additionally, the configuration can be loaded from an external read-only memory (flash ROM) into the permutation lookup table. The permutation table is a write-only memory. It can only be read out from the processor but not from the outside world. To secure the permutation against modification, a signature check can be done when loading the permutation. When installing an operating system (OS), a part of the OS runs under the identity permutation. At a certain point of the installation, the user enters a key to activate the final permutation. The activation can only be done in supervisor mode of the processor. After this point a new permutation is written to the processor. When trying to write a new permutation in user-mode the processor will signal an error. The code which is executed from this point on runs on the new permutation.

This would mean that all software, including the operating system, would have to be modified by a binary-to-binary translator (see [1] for an example) according to the inverse permutation. In order to prevent operating system changes, one could think about running the processor with the identity permutation in supervisor mode and with the permutation in user mode. Upon any syscall, the permuted encoding is switched off, and upon a return from the syscall, it is switched on again. The system could provide a database of installed software. During installation of a software, this software would be subjected to the inverse permutation, and added to the data base. In order to prevent a virus doing that, the database and translator could be password protected, so that any such action would require the system administrator to enter the password. Clearly, there are more things to be handled. Applications updates, while less frequent than on an ordinary desktop PC, will occur from time to time. Then, the affected application will have to be translated according to the permutation first, so that it returns into unmodified state. Then the update can take place, and finally, the updated application is subjected to the inverse permutation.

Another thing to consider are the parts of the operating system that run in user mode. They would have to be modified as well.

## 4. Experiments

So far, we performed only some very preliminary experiments. Instead of testing unmodified executables on a modified microprocessor, we tested modified executables on an unmodified microprocessor. To this end, we compiled a program written in C on a Linux system with the GNU C compiler `gcc`, with the `-S` option, to stop at assembler instruction level. Then, with the help of a script, we exchanged pairs of instructions, such as `subl` and `movl`, and then ran the `.s` file through assembler and linker. Next, we started the resulting executable. The original C program did nothing than opening a file, writing 1000 randomly chosen bytes, and close the file. In all instances, the execution ended with a segmentation violation. In one instance, a new file was created with 0 bytes. This may be attributed to the fact that we only changed a small number of opcodes, so that the executable may run for some time before violating the operating system protection.

To validate the scheme from Figure 1, we modeled this circuit in VHDL and implemented it into a Xilinx Virtex-E XCV1000-8 (1.8 V) FPGA [11]. To do this, we selected a global clock timing constraint of 400 MHz. Then we synthesized the circuit using high place and route effort. The constraint matched so that the circuit will effectively run at 400 MHz (internal clock). For the lookup table, we used 6 bit entries for the microprogram start and translation addresses, respectively. With 64 entries, the total size of the lookup table was $64 \cdot 2 \cdot 6 = 768$ bit. Besides address, data and write-enable, we used the supervisor flag as input. If the supervisor flag is set, no translation will be applied. If it is not set, we suppose that the processor runs in user mode and the applied address will be searched in the lookup table until a match occurs. The corresponding permuted address will be used as the start address for the microprogram. If no matching address is found in the lookup table, we signal an error. Table 2 shows the device utilization summary after place and route.

| | | |
|---|---|---|
| Number of External IOBs | 15 out of 404 | 3% |
| Number of SLICEs | 4 out of 12288 | 1% |

**Table 2. FPGA device utilization after place and route.**

To compute the dynamic power consumption, we assumed a change in the address lines with frequencies of 400, 200, 100, 50, 25, and 13 MHz (least significant to most significant bit). The power consumption was computed to be 34.7 mA, 62.45 mW. The results show that there is a great potential for the proposed circuit from Figure 1, since we did no optimizations concerning the address matching, and nevertheless reached a very high clock frequency even for an FPGA implementation. An ASIC implementation will yield much higher clock frequencies and much lower power consumption. We conclude that the performance loss due to the lookup can be minimized. Since the scheme is very simple, it can be integrated into existing microprocessors without tremendous efforts.

## 5. Conclusions

We presented a novel microarchitecture-based technique to detect and prevent malicious code execution for contemporary microprocessors. We also sketched schemes how to provide appropriately compiled applications, and how to deal with operating system issues. Further research will be more experimentally, by trying to change an FPGA-based microprocessor accordingly and install a simple web server system, which we place onto the web.

## References

[1] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. 10th ACM Conference on Computer and Communications Security (CCS)*, pages 281–289, Oct. 2003.

[2] F. Cohen. Computer viruses. *Computers & Security*, 6:22–35, 1987.

[3] Computer Economics. 2001 economic impact of malicious code attacks. http://www.computereconomics.com/cei/press/pr92101.html.

[4] D. E. Denning. An intrusion detection model. *IEEE Trans. On Software Engineering*, SE-13(2):222–232, Feb. 1987.

[5] D. Ferbrache. *A Pathology of Computer Viruses.* Springer, 1992.

[6] A. M. Fiskiran and R. B. Lee. On-chip lookup tables for fast symmetric-key encryption. In *Proc. IEEE 16th Int. Conf. Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 356–363, July 2005.

[7] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, 1991.

[8] A. Monden, A. Monsifrot, and C. Thomborson. A framework for obfuscated interpretation. In *Proc. 2nd Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation (CRPIT '04)*, pages 7–16, 2004.

[9] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an internet worm. In *Proc. 2nd Internet Measurement Workshop*, pages 273–284, Nov. 2002.

[10] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *Proc. 2004 ACM Workshop on Rapid Malcode (WORM '04)*, pages 33–42, Oct. 2004.

[11] Xilinx. Virtex-E 1.8 V field programmable gate arrays, 2002. http://direct.xilinx.com/bvdocs/publications/ds022.pdf.

[12] X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. *ACM SIGPLAN Notices*, 39(11):72–84, 2004.