Embedded System Design Seminar

Seminar Report

Secure Processor Design

Filipe G S Valentim

Suprvised By

Prof. Dr. Rainer Leupers

Chair for Software for Systems on Silicon (SSS)

Institute For Communication Technologies And Embedded Systems (ICE)

RWTH Aachen

Contents

Chapter 1

Introduction

1.1 Abstract

In this work, the basics to understand a hacker attack on a processor is laid, with the ultimate aim to expose two new proposals of secure processor design. This is done by laying first some common processor design techniques, then how the hackers use this features in their advantage. After citing some important and recent kinds of attacks, the suggestions for the secure processor design are exposed.

1.2 Introduction

Even with decades of research, software and hardware systems still have openings to attacks, be it memory corruption of other kinds of exploits. This comes from a diversity of factors, but the hackers rely on the architecture of the system itself to attack the memory. The major exploits are laid in chapter 3 of this work, the central being the attacks over the Address Space Layout Randomization (ASLR), that is still susceptible to Return Oriented Programming (ROP) and Data Oriented Programming (DOP) attacks. To understand that, it is necessary to first understand some of the architectural elements of the processor itself, so these are first exposed in chapter 2. It is important to see the complexity of the system, and how every part of it must be designed with security in mind. The TCB (Trusted Computing Base) is a central concept to this topic, and is exposed also in chapter 2. Due to the importance of a secure system, lots of new proposals to increase the security of processors are being published, and two are exposed here, Smokestack and Morpheus. Both of them use randomization techniques to render the attacker's effort useless (Chapter 4). After that, the conclusion summarizes the most important topics presented.

Chapter 2

The Processor Design

2.1 Definition - What is a processor?

A processor is the main integrated circuit chip (IC) in a computer. It is responsible for reading and interpreting most of the computer commands. It can process basic arithmetic, Input/Output and logic operations, as well as the management and allocation of commands for other chips and components running in the system.

As of its crucial and central nature in the computer design, one must have total reliance on its security, as a fault in the computing of a car engine or in bank operations could lead to disastrous consequences. In the electrical design done today, processors and microprocessors (the ones that have embedded everything they need to run inside themselves) are almost everywhere, so much that naming their uses is pointless.

2.1.1 The basic elements of a processor

The basic elements of a processor include:

ALU - Arithmetic Logic Unit

Computes the arithmetical and logical operations on given operands.

CU - Control Unit

Circuits that fetch an instruction from memory, decode it and pass the instruction to ALU, Registers, RAM or I/O.

Registers

Memory storage circuitry, often named as immediate memory and assist ALU and CU in different operations.

2.1.2 CPU Operations

The primary operations of a processor are:

Fetch - Step in which the program counter indicates the next instruction that needs to be read from the memory.

Decode - Converts the instruction to understand which components are needed to execute the operation.

Execute - The actual action is performed in the specific areas of the CPU, decoded in the last step.

The instruction set of a processor is the list of instructions that are able to be processed. The Instruction Set Architecture is the guide to some hardware design decisions, so it must be decided in the beginning of the design process.

2.1.3 How the code runs

In order to run, the code is passed through the toolchain. The toolchain is a system of four parts: compiler, linker, assembler and debugger. It is a set of programs that take the high-level code (i.e, code written in C++, C, Java) and transform it in a binary file (machine code) that is executable by the processor.

2.2 Processor Design

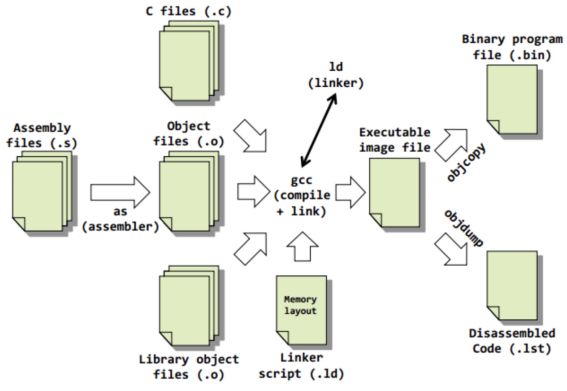


Figure 2.1: The toolchain steps. Source: `https://www.bogotobogo.com/cplusplus/embeddedSystemsProgramming_gnu_toolchain_ARM_cross_compiler.php`

Processors are intricate systems of trillions of transistors, interconnection layers and wires. The design of a processor is not simple, and involves multiple companies, with hundreds of people of different backgrounds.

As the throughput of the machine is key, the designer must carefully choose the components and instructions allowed in the system. The Kernel is the most trusted party by all systems with rings outside it (see figure 2.2), as the hardware is the most privileged, for being the lowest level in the system. It forms the *Trusted Computing Base - TCB*, with the systems that

are supposed to realize the protection effort of the system.

Key to most secure processor architecture designs is the trusted processor chip assumption, in which one assumes that the memory, the busses and the devices are not to be trusted. Also, Kerckhoffs's Principle from cryptography can be applied to secure architectures, as the system must remain safe if everything is known, except its cryptographic keys.

2.2.1 Design Techniques

Some important design techniques of modern CPUs are:

Privilege levels

For every process in the processor, there is a domain named protection ring. The access permissions for an inner ring are bigger than those to the rings around it (see Figure 2.2).

This set of permissions form what can be called nested subsets. For each segment of memory, there is a permission flag that indicates if it is possible to read, write, or execute the process. A process must be able to interact with a segment in its memory only if the ring of execution of the process has the right permission. [11]

Memory Mapping

A system that maps a program file on disk to a range of addresses within an application's address space, emulating dynamic memory, as it can now be accessed the same way. This speeds up the reading and writing procedure.

Instruction pipelining

This technique implements instruction-level parallelism in processors that are single core, keeping the full system busy by dividing the instructions into sequential steps. These are then realized by different units, as different parts are processed in parallel. The ultimate aim is to compute an instruction on each clock cycle. However, when a subtle change of instructions occurs, the pipeline sometimes must discard the current process and restart. This is called a "stall."

CPU cache

Cache memories are systems that are physically close to the cores to hold information relevant to the processing. It increases the execution rate, because the access time of the cache memory is much lower then the main memory.

Dynamic Voltage and Frequency Scaling (DVFS)

This system watches the dynamic voltage and frequency scaling to the change in resource utilization of the device. If there is a big deviation from a threshold amount, the clock frequency or the voltage island is adjusted. One way to measure resource utilization is the number of instructions executed per second. If resource utilization has increased, the frequency/voltage increases, as the adjustment depends on preprogrammed operating points that relate to a power management state.

Out-of-order execution

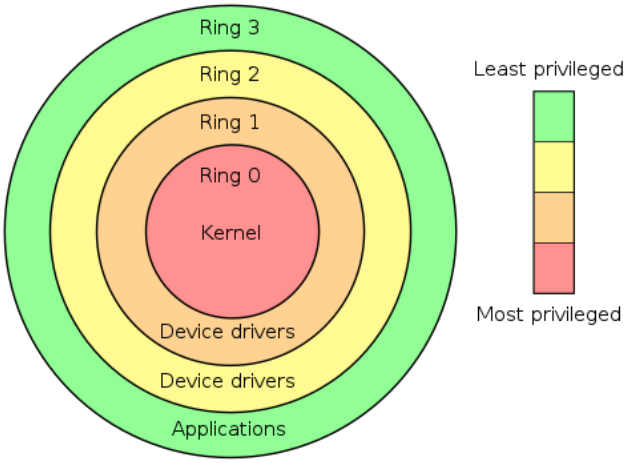
For maximizing the use of processing units of the CPU core, it is usual to let other executions units run ahead of the sequential program order, so the CPU can execute the instructions only when all resources are available. Therefore, it is possible to run instructions in parallel with the architectural definition constraint.

Branch prediction

With out-of-order execution and instruction parallelism, the system has to handle the fetch latency limitation. To overcome this issue, branch prediction emerges as a form for prefetching, when the processor tries to predict the result of a branch instruction so that it can speculatively fetch the needed instructions. Then, the processor executes the instructions, hoping the result is going to be useful and the prediction is correct. The gains come when the successful prediction rates are higher than the misprediction penalties. This way the overall performance can be better. If this system didn't exist, the processor would stall whenever there were unresolved branch instructions. The average of branches in code is 20 percent. [10]

2.3 Security: a major concern

Figure 2.2: The privilege level (protection rings). Source: `https://commons.wikimedia.org/wiki/File:Priv_rings.svg`



Now that the major steps and philosophies in the design and execution of processors are laid, it is possible to have a preview on the complexity of this topic, as for understanding the secure design, one must understand the attacks (threat model), and for understanding the attacks, one must understand some design structures. A lot of different companies, in different countries, are part of these steps. In the next chapter, we are going to see some common attack techniques

Chapter 3

Attacks

3.1 What is an attack?

The general philosophy behind an attack is to take actions directed against computer systems to disrupt equipment operations, change process control, or corrupt stored data. Data leakage is pretty common, followed by control of operations. It is usually important to the hackers to hide their traces, keeping the attack covered. The worst type of attack is the one who achieves Turing-complete operations, in which the hacker has total control over the system (as the ability of changing memory variables) without the operator knowing.

Privilege Escalation

The general goal of an attack is privilege escalation, when the hacker can control a secure and more privileged system by the means of another less privileged one. For this, exploits are used, which are potential problems in the system, such as a bug, design flaw or configuration oversight in an application. That way, the hacker can gain elevated access to protected resources, performing unauthorized actions. (See Figure 2.3)

3.2 Threat Mode: Hardware approach

Some attacks could happen during the manufacturing of a chip, as the IC passes through different companies in the supply chain. Not necessarily they are avoidable, and some of them pose a bigger risk to the company's business than to one's data itself. The possibilities are infinite here.
There are some hardware attacks, however, that require no physical access. Some of them are:

Rowhammer

By repeatedly accessing the same address in Dynamic RAM, it is possible that the adjacent addresses get corrupt, via electromagnetic interaction.That is, by reading and writing to an address in DRAM, others addresses get corrupted, in nearby rows. It was demonstrated on Intel and AMD hardware using a program that forces a lot of DRAM accesses. [3]

Meltdown

By using the side effects of out-of-order executions, Meltdown can read arbitrary kernel-memory addresses, including personal data and passwords. This breaks the security features provided by address space isolation, and thus, every security mechanism built upon these assumptions. Meltdown technique is capable of reading in sequence every instance of cache memory. This could lead to the leakage of sensitive information from the processes that lay in the memory. [4]

Spectre

The attack forces the performance of abnormal operations that leak confidential data. For that, it uses a combination of techniques in such a way that it is possible to read random memory addresses from the process in execution. Speculative execution implementations overcome the security systems by opposing the basic security assumptions. These assumptions include operating system process separation, containerization, and just-in-time (JIT) compilation. [5]

3.3 Threat Model: Software approach

These are the most common type of attacks, therefore an infinity of viruses and malicious exist. They all use some exploits, such as:

Memory Corruption

Memory Corruption is characterized when the content or the pointers in memory are modified unintentionally. This can be caused by behavior that exceeds the intention of the original program and can also be termed as violating memory safety. If it is caused by an attack and if the corrupted memory contents are successfully executed, it could lead to a successful attack.

Control Flow

The control flow system operates the program counter (PC). The PC is a pointer that increments after fetching an instruction, and holds the memory address of the next executable instruction. The fetching of instructions is normally sequential, however, control transfer instructions can change the sequence, by replacing the value in the PC. In attacks that use this, the execution flow of a program is manipulated to execute code sequences not anticipated by the programmer, with the

ultimate goal of circumventing system security measures. The attacker uses memory errors to corrupt control data, such as a function pointer, return address, or C++ virtual function table, to eventually hijack the execution flow of the program.

Data Oriented Programming Attacks

Data-oriented programming attacks try to contaminate non-control data to execute batches of instructions within the program. This contaminated piece of software now has attacker-controlled operands. These batches of instructions are called a DOP gadget, and they are just a small part of the overall attack. It is possible to obtain Turing completeness by linking one DOP gadget to another in a loop, which is called a DOP gadget dispatcher.

Defeating Address Space Layout Randomization (ASLR)

ASLR is an algorithm that assigns random memory space addresses to a program component. It is meant to be used in a OS to prevent or lower the risk of vulnerabilities being used by third parties for memory attacks, such as buffer overflow. It prevents the direct access to memory locations, making more difficult to ascertain the position of a program's specific block in RAM, so that the attacker cannot reuse existing code. Kernel Address Space Layout Randomization (KASLR) is one of the most adopted techniques to avoid corruption susceptibilities, as buffer overflow and use-after-free. For it to be secure, there must be no memory leakage and high entropy. [6]

3.3.1 Examples

With the common exploits put above, we now look at some new attacks that a secure system must account for.

Blind Return Oriented Programming (BROP)

As stated in [7], even without knowing the target binary or source code in advance, it is possible to use stack buffer overflow attacks, specially for services that need to restart after a crash. With this, it is possible to hack open-source servers, or even proprietary closed-binary services that have binary code unknown to the hacker. Usually, the hacker disposes of some binary code used in a specific distribution of the target software, in which Return Oriented Programming (ROP) is attempted. However, the Blind ROP (BROP) tries to gather ROP gadgets in the program to perform a write system call, then it transfers the malicious code to the system, in which another already known technique can be used.

Crash Resistant Oriented Programming (CROP)

Some client applications continue to run after a crash and accept errors which are usually critical and would trigger termination. To these, a crash-resistance primitive and a memory scanning method can be used, even without the need for control-flow hijacking. It is demonstrated by [8] that memory corruption vulnerabilities can lead to an alternative way of weakening the security features of ASLR.

JUMP over ASLR

This attack tries to input an offset in kernel and user-level ASLR, by the means of a side-channel attack on the branch prediction system (Branch Target Buffer - BTB). The attack uses BTB collisions during the executions of instructions of the virus and the local processes of the victim, being them user-level or kernel. This effect influence the virus' code timing, grating the attacker to know the addresses of branch instructions. Some versions of this method recovered kernel data in 60 milliseconds in actual versions of Linux. [9]

Chapter 4

Secure Processor Design

In this chapter, two approaches for the improvement of security in processors are laid. The first is a hardware approach, that changes some units inside the core of the processor; and the second is a software approach, that can be made to run in any machine.

4.1 Morpheus

Designed by [2] in 2019, Morpheus focus is mainly to mitigate control-flow attacks (see Chapter 3.3) in hardware level, thwarting attacks that forces the victim to process untrusty corrupt data. The attacks would use the exploits handled in chapter 3 to override protections and hijack control flow, but fail due to the continuous obfuscation of information needed to perform the attack. The goal of the design was a *"vulnerability-agnostic secure system"*, when the computing platform behaves normal to most programs but hostile to malicious ones.

Limitation: The work does not comprehend Denial-of-service (DoS) and side-channel attacks.

The team based their work in some general assumptions about the attacker, the two most important being that: it is not a physical attack and it is possible to locate memory corruption or vulnerability in the target program. The TCB should include Morpheus special hardware and software.

The work combines two methods of protection: randomization of key values and churn.

The randomization of values works relocating and encrypting pointers and code, and are described by: *"Ensembles of moving target defenses"* (EMTD). This ends up forcing the attacker to examine the system before the attack, and it is easier to force the probation of the system to fail. Then enters the churn, which re-randomize program values while the system is still running, forcing the attacker to reprobe the system every time churn happens, locking the attack in the probing phase. This is used for code, code pointers

and data pointers.

4.1.1 The Central Mechanism

Central to this architecture, the EMTD rely on tagging to keep track of all memory objects during runtime. Morpheus implements the EMTD is two manners: pointer displacement and domain encryption. The first secures pointer values by adding a random offset to their location, and the latter generates a layer of encryption to all domains of the program. The churn unit has the ability of re-randomizing these during runtime, while still keeping efficiency. Also, when the systems detect an attack, the churn rate is increased.

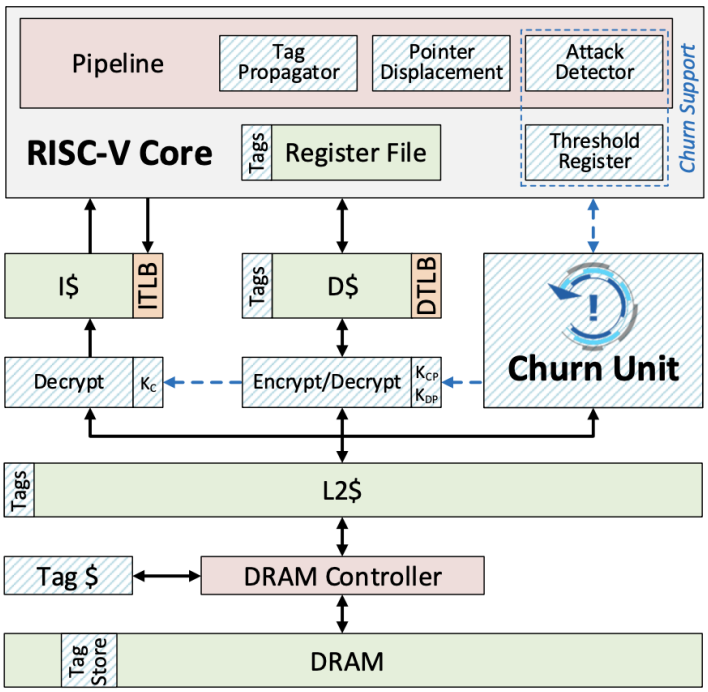


Figure 4.1: The Morpheus Secure Architecture. *"Components hashed with diagonal lines augment the baseline RISC-V system to support Morpheus defenses. The dotted line is a bus used for churn control signals and transmitting keys. Churn Support includes the Attack Detector and Threshold Register, while logic for propagating tags and translating pointers is added to the pipeline."* Taken from [2]

When combined, these protections show a increase in the protection against control-flow attacks, while having very low overheads. This new way of displacing pointers increases the entropy of DAS displacement (with 60-bits) significantly more than ASLR-based defenses, as it uses normally 30-bits. It also increases the difficulty for a forgery or leakage of information, as the new pointers can only be derived from knowing an existing pointer value.

The system is believed to be more effective even with the evolution of attacks (or future-proof), because it defends the assets that are critical to the attack.

4.2 Smokestack

Designed by [1] in 2019, Smokestack focus is a software level, runtime **"stack-layout randomization scheme that can effectively thwart DOP attacks"**. To achieve that, the system randomly permutes the local stack for every function that is called. The system also utilizes a true-random number generator, that is protected against memory attacks, and makes the permutation untractable.

While the state of the art randomization techniques in code have problems stopping DOP attacks, Smokestack excels at stopping said attacks with the randomization of the stack frames at every call of the functions, causing a minimal delay in the execution.

Some assumptions about the attacker were made, and the main ones are that it is possible to read/write to all data memory (except for the non-writable registers) and the brute-force attack has a limited time before detection (the service would restart then).

4.2.1 The Central Mechanism

Smokestack attempts to allocate stack frames with on the run re-randomization. However, it still keeps all the good features of stack allocation (e.g. easy deallocation of objects). For this, the system first chooses an index (based on the true-random number generator) and then replaces a piece of the total stack allocation with this index. To increase the overall performance of runtime permutation, it has a special program just for that: the read-only permutation box, P-BOX. It works as a lookup table with all the possible permutations of all unique stack frames. For the protection to render the attacker's efforts useless, the variable's absolute address must change every function call, as well as the relative distance from the other state parameters. This is achieved during what is called prologue: when the system decides the parameters relative to the randomization. During the call of a function, a random permutation from the P-BOX is chosen based on the output of the random number-generator. Then, the system reallocates the stack frame and assigns their addresses of memory based on the output of the P-BOX.

The true-random selection of permutations is impossible to be predicted, even by attackers that have write/read permissions to the data memory. The overall performance is dependent on the algorithm used to generate the random number. Smokestack is specially designed for preventing DOP attacks, and by randomizing stack frames for functions, it ensures that the needed information for a DOP gadget attack is unaccessible for the attacker, as the absolute address and the relative distance to a stack object are unpredictable for the function call performed. Smokestack has shown low memory overhead on applications, and it is modular to support migration of code.

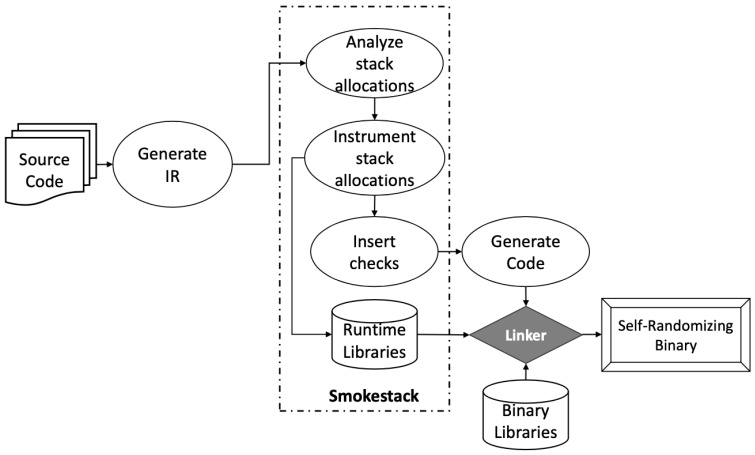


Figure 4.2: Smokestack overview. Taken from [1]

Chapter 5

Conclusion

After explaining some basics about some security and efficiency measures of the modern processors, as well as some exploits used to attack these measures, this work shows two new proposals that are similar in nature to prevent new attacks. Both Smokestack and Morpheus use randomization of pointers and memory mapping to render the attacker's effort useless. This is a better implementation, as while traditional protections try to find and protect every single vulnerability, these mitigate the attack as they hide the information needed by the attacker. Both of these implementations showed better protecting results than traditional ones. While keeping the efficiency in mind, these new techniques protect the main system from control attacks, utilizing hardware and software to achieve that. For the next generation of security systems, these implementations should be carefully studied, as they showed great efficiency, increase of security and future-proofing. The main philosophy behind the future security techniques in this line is to explore and somehow hide the assets needed by the attackers, so they can no longer develop efficient mechanisms and viruses.

References

[1] Misiker Tadesse Aga and Todd Austin. *Smokestack: Thwarting DOP Attacks with Runtime Stack Layout Randomization*, 2019. DOI: https://dl.acm.org/doi/10.5555/3314872.3314879

[2] Mark Gallagher, Lauren Biernacki et al. *Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn*, 2019. DOI: https://doi.org/10.1145/3297858.3304037

[3] Yoongu Kim, Ross Daly, et al. *Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors.* 2014. DOI: https://doi.org/10.1145/2678373.2665726

[4] Moritz Lipp and Michael Schwarz et al *Meltdown: Reading Kernel Memory from User Space*, 2018 https://meltdownattack.com/meltdown.pdf

[5] Paul Kocher and Jann Horn et al, *Spectre Attacks: Exploiting Speculative Execution*, 2019 https://spectreattack.com/spectre.pdf

[6] Yeongjin Jang, Sangho Lee, and Taesoo Kim. *Breaking Kernel Address Space Layout Randomization with Intel TSX.*, 2016 DOI: https://doi.org/10.1145/2976749.2978321

[7] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières and D. Boneh, *Hacking Blind*, 2014 DOI: 10.1109/SP.2014.22.

[8] Gawlik, Robert; Kollenda, Benjamin; Koppe, Philipp; Garmany, Behrad; Holz, Thorsten. *Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding*, 2016 DOI: 10.14722/ndss.2016.23262.

[9] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. *Jump over ASLR: attacking branch predictors to bypass ASLR*, 2016, https://dl.acm.org/doi/10.5555/3195638.3195686

[10] Ben Lee, *Dynamic Branch Prediction*, http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/#l0

[11] Michael D. Schroeder and Jerome H. Saltzer. *A hardware architecture for implementing protection rings.* 1972, DOI: https://doi.org/10.1145/361268.361275