2010 International Conference on Availability, Reliability and Security

A computer architecture with hardwarebased malware detection

Klaus Hildebrandt
Helmut-Schmidt-University
Holstenhofweg 85
22043 Hamburg, Germany
klaus.hildebrandt@hsu-hh.de

Igor Podebrad
Commerzbank AG
Mainzer Landstr. 193
60261 Frankfurt am Main, Germany
igor.podebrad@commerzbank.com

Bernd Klauer
Helmut-Schmidt-University
Holstenhofweg 85
22043 Hamburg, Germany
bernd.klauer@hsu-hh.de

***Index Terms*—secure computer architecture, malware, hardware based security, multicore**

***Abstract*—In the past the computer architetures and technologies have only been optimized for speed - never for security. The aspects of IT-security were always disregarded. [1] shows that it is possible to write malware that can not be detected or removed from an infected system by software. *Stealth*-features are possible by special properties of the x86 architecture. This work describes a general hardware architecture, that will improve the security.**

INTRODUCTION

According to [2], the amount of computers will still grow in the next years. Not only the number of desktop computers and large computer systems will increase but the number of embedded systems as well. The microcomputer will significantly influence everyday life. They will be used for human health monitoring, in the traffic managment or in aerial/space technologies. The dependency of single persons and the society on these systems will definitely grow. Today almost all digital attacks are directed against personal computer and mainframes/server. The embedded systems in the sense of microcontrollers have not yet been discovered as an attack target. This will change with the interlinking of microcontrollers. Trends in this direction are outlined in [2] and [3]. Car-to-car-communication and ambient assisted living are upcoming usage areas of ”invisible” microcontrollers. A preview of such attacks on microcontrollers can be seen in [4]. This atricle describes how home routers can be converted to bot nets. The meaning of the ”invisible” computers increases and their active and passive damage potential increases as well.

Today IT security research is focused on desktop and desktop-like computers. For example the co-pilot’s project [5] monitors the integrity of Linux kernels assisted by a PCI card. However, such technologies are only applicable with certain computer families. The need for uniform methods and structures to improve security of several computer systems will definitely grow in the future. This work describes a general hardware architecture, that will improve the security.

I. MALWARE TAXONOMY

[1] defines malware as *a piece of code which changes the behavior of either the operating system kernel or some security sensitive applications*. The life cycle of malware can be splitted into three parts:

- infection
- spreading
- execution of malicious actions

During the spreading phase malware migrate to other systems. This new intallation is in the infection phase again. The changes during infection occur without permissions from a user or the operating system. In most cases it is not possible to detect or prohibit these actions with tools of the operating system. After the infection phase malware must be executed to initiate the spreading phase or to do harmful actions. Although spreading is already a harmful action it is considered separatly from the harm phase here. From the destruction of data to spying and up to the acquisition of false pretenses of computing performance, varying damage effects were observed in the past. Modern age malicous code structures orientate on modern software engineering techniques: downloading of updates make the damage effect arbitrarily interchangeable — and unpredictable.
To activate the described techniques, malware must be executed by a processor. Malware that will not be executed is harmless. In 2006 Rudkovska has introduced a malware taxonomy, that is based on the type of activation of the malware. We will introduce this taxonomy in detail as it is a key for successful malware protection.

A. Malware taxonomy in depth

Rutkovska’s malware taxonomy ([1]) classifies maleware by few but very important aspects of system architecture. In this taxonomy four types of malware exist. The simplest type is *type 0*-malware. This type doesn’t change the kernel nor the applications. It uses security holes to spread itself. Normally a type 0-malware is a simple process in the system. The harm of this type is unbounded(figure 1). The type 0 has three subtypes:

1) a program that has no tasks, except its spreading and malicious actions. This type of malware will be recognized by anti-malware-programm by it’s signature. These programs don’t try to hide themselves.
2) a useful program that was infected by malware. This

978-0-7695-3965-2/10 $26.00 © 2010 IEEE
DOI 10.1109/ARES.2010.39

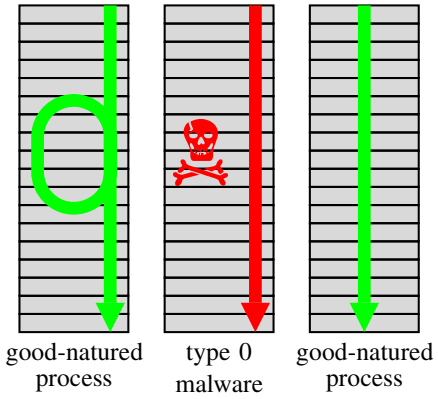

Fig. 1. Operating mode of type 0 malware

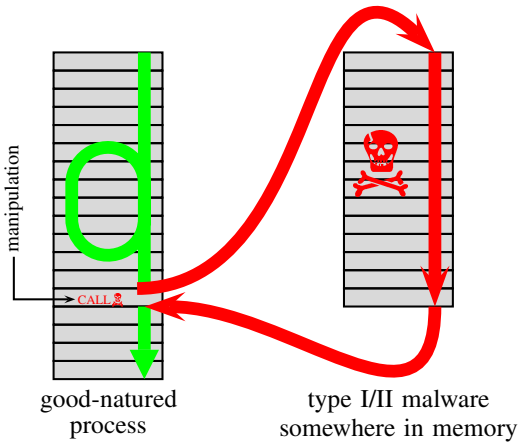

Fig. 2. operating mode of type I and type II malware

kind of malware can be detected by anti-X-software[1] as well.

3) a not infected or changed program from a wellknown and trustable manufacturer but with additional, hidden functionality. Some well known programs have picked usage-data and sent it home. Another example is Sony-BGM-Rootkit([6]).

Type I-malware changes memory areas that must be unchangeable by definition (by not by enforcement). For example *in-memory-code-sections* in application and OS-kernel should no be changed. A software that rewrites its own or foreign code sections is a malware of type I. In most cases the change of instructions in memory works like the example below:

1) Load the code with malicious functionality into memory. This is often camouflaged as data loading.
2) Change instruction(s) of "good" process(es) to put a previously loaded segment into the execution path to execute a malware in the context of the host process.

The simplest kind of change is insertion of a `CALL`-instruction into the host process (look figure 2). Because the malware will be executed in the context of the infected process, it inherits the permissions of the host process.

Type II-malware changes only recources that are allowed to be changed. For example: in-memory-data-section in processes and kernel. A classic features of von-Neumann-architecture is used by type II-malware: the same memory is used for instruction and data. So data can be executed and instructions can be overwritten. The difference between instruction and data in von-Neumann-architectures is the semantic only. A valid instruction sequence can be executed from data segmant too. The only prerequisite is a correct JMP command only. Another possible way for type II-malware is to recurve a function pointer to jump into malicious code. Note: pointers to functions are data, not instructions. But with the help of this pointer the execution path can do jumps, too.

Type III-malware exists outside of an operating system. It changes nothing in the kernel or in the applications. Typ III-malware use virtualization techniques like Intels Vanderpool or AMD's Pacifica to hide itself. The infected system is not running directly on hardware but in a virtual machine supervised by the malware. An example for a type III-malware is BluePill([7]).

[1] anti virus/anti adware programs

II. Hardwarebased techniques to detect and remove malware

The execution of type-0 malware can be prevented by using of digitally-signed executables like [8].

A. Hardwarebased intergrity protection of instructions

A system with integrity of instruction is type-I-resistent.

The operation of type-I malware is possible because of a security hole in the von-Neumann architecture (Fig. 3): the usage of the same memory for instruction and data. Data and instruction are distinguished only by interpretation, because a simple memory instruction can be overwritten camouflaged as data storage. Techniques like NX-Bit or W^X-bit try to split memory into instruction and data areas and check that the target of the storage operation is only a data area. These techniques require a tight cooperation of the hardware and the operating system. But operating systems can be hooked by security holes and the interplay will be corrupted. The separation of instructions and data on the physical level increases security, because silicon can not be changed by software. An architecture providing this feature is the Harvard architecture. The Harvard architecture has *two* memories: one for instructions and one for data(Fig. II-A). Every memory is connected to the processor via a separate bus. The original goal of the Harvard architecture was to increase the throughput the the CPU/memory bus. The resistance for type-I malware by integrity of instructions can be reached by making the instruction memory readonly accessable for the processor. From the perspective of the CPU, the instruction memory is a ROM. To hold the universality of the system, a possibility to load programs must be given.

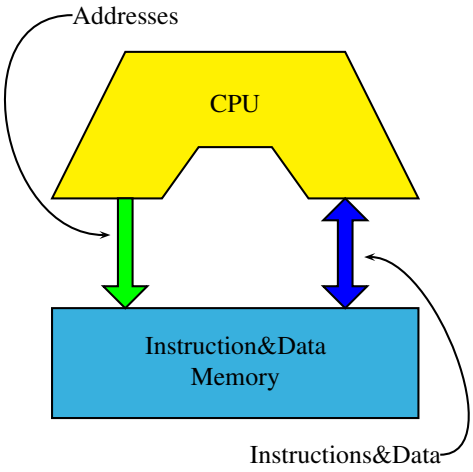

Fig. 3. von-Neumann architecture

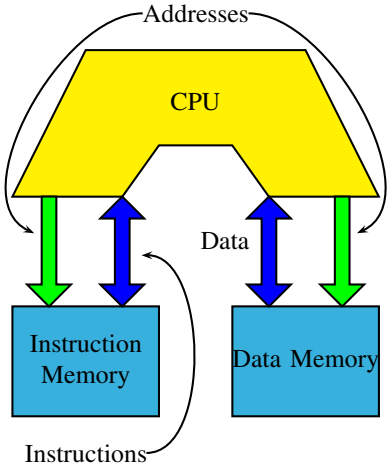

Fig. 4. Harvard architecture

The changes in the instruction memory can not be made by a running program directly, because in this case the integrity of the commands is lost again. One idea of this work is to manage the instruction memory by a special security processor (chapter III). The architecture with instruction integrity can be seen in Fig. 5. This architecture cuts the type-I malware livelyhoods, as the instruction memory is not longer writable by malware execution. An exclusive way to set or modify instructions in memory is via a security processor.
The architecture as described above reduces the universality of the von-Neumann atchitecture by the feature of self modifiying code on one hand. On the other hand it introduces the security features of non-executable data and instruction integrity. Non-executable data and instruction integrity are essential for a secure computer architecture by [9].

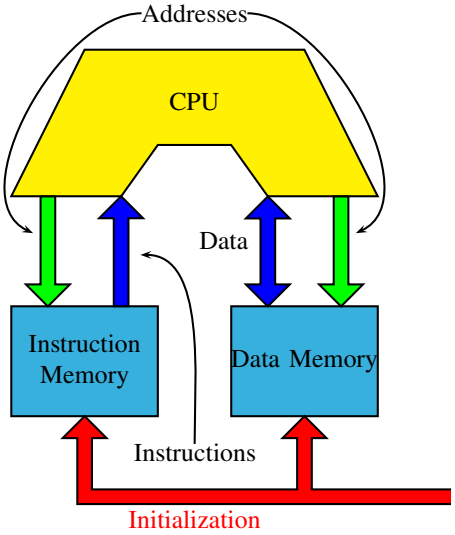

Fig. 5. Security-Harvard architecture

B. Hardwarebased recognition of type II-malware

The usage of an architecture described in chapter II-A makes the execution of type-II malware difficult because the instructions can be received from instructions memory only. The storage of the instructions in data-memory is possible, but these ”instructions” cannot be executed. This requires a transfer in the instruction memory. The transfer can be performed only be performed by the security processor. This transfer reduces the classification of the attack to type-I (see above). Such a transport enforces check on the security core. A security core has the potential to identify harmful programs/code. This will exclude malicious code from execution.

a) Instruction address monitoring: Programs for the secure Harvard architecture (Fig. 5) can be stored in a simple continguous memory segment. Manipulation of instructions at runtime is only possible via the security core. The security core executes an operating system to control the whole system and has mechanisms for integrity- and authenticity-checks. A security hole is the linking of malicious code into the execution path over the change of a funtion pointer. Function pointers are data in data memory but contain addresses of instructions. A modification of data memory can cause a jump into foreign code. Such jumps can be detected with hardware-based *address monitoring*[2]. On every instruction access an Addressguard can check if the requested address of an instruction meets a specific memory segment. A segment violation will be reported to the security core and causes countermeasures if necessary. This procedure works fine for statically linked programs. In the next chapter we will show how it can be adapted to dynamically linked programs.

b) Data address monitoring: By applying the Adress-Guard to the data bus it is possible to log and analyse access

[2]software-based monitoring is too slow!

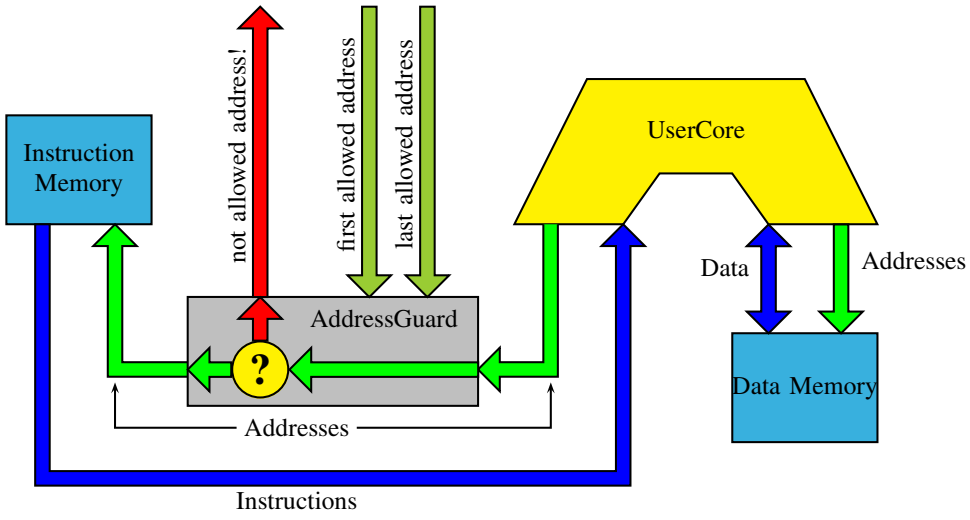

Fig. 6. An architecture to detect not permitted jumps

on the data at run time. As a consequence unauthorized access can be detected by the security core and can be effectively prevented.

III. The secure IT architecture

The described secure computer architecture has two essential features:

- separation of user- and application memory space from OS/kernel space
- both the *user core* and the *security core* are implemented as cores decribed in above sections like Fig. 5.

The architecture consists of at least two cores and the memory is physically splitted in the user and kernel space. Both cores are implemented as security Harvard to be resisten for type-I malware. The security core hosts operating system and security applications only. Due to the security instruction memory of security core is a ROM. The main tasks of the security core and the OS are to (re)load the programs in the user memories (instructions and initial data) and to deal with address violations on the user core. A (re)loading of programs to user space must contain a check of the digital signatur for example like [8].

The user core is available for user. It can executes every non-self modified program and is a "worker" in a this architecture. By using of an address guard from chapter II-B every jump from loaded instruction segment will be repoerted to security core/OS an can be handled. But on systems that allow dynamically-linked binaries not every reported address violation is a jump into foreign code. It can be a jump into another segment of a binary file. Dynamically linked libraries must not be located at contiguous address areas. The OS on security core can store the bounds of all segments for every process and can allow jumps to other code segments[3]. This make the execution of dynamic linked libraries still possible.

IV. Implementation/Proof-of-concepts

A. Implementation

To perform investigation of the concepts as described above a Harvard-processor with an ARM-instruction set was implemented in VHDL and synthesized on a Xilinx FPGA. The usage of the ARM instruction set ([10]) for CPU impementation saves the compatibility to legacy code[4] and permitts usage of usual compilers and tools for software development.

The above described and in Fig. 7 presented system was implemented on Vertex5 XC5VLX50T FPGA and 75 % of FPGA area was used. A minimal operating system was written. As communication interface an RS232-conntector was used. The operating system can recieve an ELF file over RS232 and starts it on user core. To produce executable code a GNUARM compiler was used.

The executability of such legacy code is given as far as the code is not self-modifing. A disadvantage of legacy code in the presented architecture is the inefficient memory usage, due to gaps in the instruction memory at former data addresses can not be used. The usage of memory can be easily fixed by the compiler. It may use identical address ranges for both code and for data without collisions. The unused areas in the legacy code can be filled with instruction that trap an interrupt. This procedure makes a detection of data execution possible.

B. Tested features

A simulated type-I attack has change some data on data memory, but a manipulation of instruction was not possible.

[3]Under assumption: one shared library is equal to one code segment

[4]Legacy code means "old" programs, that must be run on updated hardware or on new operating systems.

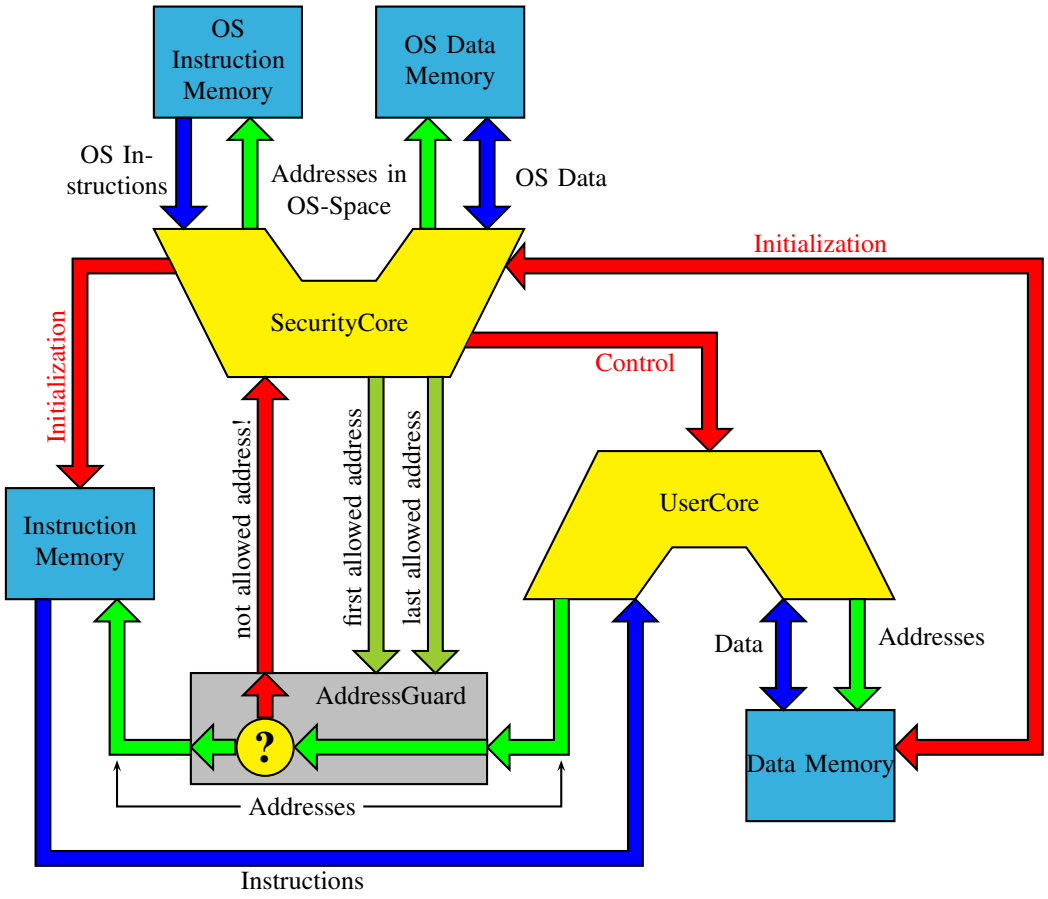

Fig. 7. Complete system to detect and remove malware

Furthermore a simulated jump injection into execution path over function pointer manipulation was detected and reported to security core/OS.

C. Future work

The authors have a strong assumption that a price for security is the performance. Future work is to look, which performance related components of a hardware architecture are conflicted with security requirement. If some performance related components are insecure und must be waived out, so way to increase performance without loosing of security must be found.

References

[1] J. Rutkowska, "Introducing stealth malware taxonomy," http://www.invisiblethings.org/papers/malware-taxonomy.pdf, Nov. 2006.

[2] T. Ungerer, "Keynote: Grand challenges of computer engineering," in *Architecture of Computing Systems ARCS 2008*, ser. Lecture Notes in Computer Science, vol. 4934. Springer Berlin / Heidelberg, 2008. [Online]. Available: http://www.springerlink.com/content/y6747q2381361658/

[3] R. Anderson, *Security Engineering*. John Wiley & Sons, 2008. [Online]. Available: http://www.cl.cam.ac.uk/~rja14/book.html

[4] heise online, "Bot-netz aus heimnetz-routern," http://www.heise.de/newsticker/Bot-Netz-aus-heimroutern--/meldung/134992, 23.03.2009.

[5] N. L. Petroni, J. T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," www.usenix.org/events/sec04/tech/full_papers/petroni/petroni_html/main.html, 02.04.2009.

[6] heise online, "Sony bmgs kopierschutz mit rootkit-funktionen," http://www.heise.de/newsticker/Sony-BMGs-Kopierschutz-mit-Rootkit-Funktionen--/meldung/65602, 01.11.2005.

[7] J. Rutkowska, "Virtualization - the other side of the coin," http://www.invisiblethings.org/papers/NLUUG-virtualization.ppt, May 2007.

[8] L. Catuogno and I. Visconti, "A Format-Independent Architecture for run-time integrity checking of executable code," in *Security in Communication Networks*, ser. Lecture Notes in Computer Science, vol. 2576/2003. Springer Berlin / Heidelberg, 2003, pp. 219–233. [Online]. Available: http://www.springerlink.com/content/73qyp53g9hlxp27a/

[9] I. Podebrad, K. Hildebrandt, and B. Klauer, "List of criteria for a secure computer architecture," *Emerging Security Information, Systems, and Technologies, The International Conference on*, vol. 0, pp. 76–80, 2009.

[10] ARM, "Arm7 data sheet," http://users.ece.utexas.edu/~chung/vlsi1/lab3/lab3b/ARM7vC.pdf, Dec 1994.