A Bytecode Interpreter for Secure Program Execution in Untrusted Main Memory

Maximilian Seitzer, Michael Gruhn(✉), and Tilo Müller

Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg, Erlangen, Germany
{maximilian.seitzer,michael.gruhn,tilo.mueller}@fau.de

Abstract. Physical access to a system allows attackers to read out RAM through cold boot and DMA attacks. Thus far, counter measures protect only against attacks targeting disk encryption keys, while the remaining memory content is left vulnerable. We present a bytecode interpreter that protects code and data of programs against memory attacks by executing them without using RAM for sensitive content. Any program content within memory is encrypted, for which the interpreter utilizes TRESOR [1], a cold boot resistant implementation of the AES cipher. The interpreter was developed as a Linux kernel module, taking advantage of the CPU instruction sets AVX for additional registers, and AES-NI for fast encryption. We show that the interpreter is secure against memory attacks, and that the overall performance is only a factor of 4 times slower than the performance of Python. Moreover, the performance penalty is mostly induced by the encryption.

Keywords: Coldboot · Secure computation · Encrypted bytecode

1 Introduction

Physical security has often been a weak point in the defense of computer systems, especially mobile ones. Against physical access, software protection methods are often no longer effective. Even though methods such as full disk encryption can protect parts of the system, namely the hard disk, encryption keys still reside in RAM. As it stands, encryption is not applied to RAM, which makes memory attacks feasible today. A memory attack is a physical attack that lets an adversary obtain a memory contents of the targeted running system. One type of memory attack is known as the *cold boot attack* [2,3]. Cold boot attacks exploit the *data remanence effect* [4] which says that data in RAM gradually fades away and can be accessed for a short period of time after powering off [5–7]. Another threat are *DMA attacks*. DMA attacks exploit the fact that *direct memory access* allows external devices to directly interface with RAM, without the operating system being involved [8,9].

G. Pernul et al. (Eds.): ESORICS 2015, Part II, LNCS 9327, pp. 376–395, 2015.
DOI: 10.1007/978-3-319-24177-7_19

1.1 Motivation

As the spread of full disk encryption extends, and devices become more and more mobile, the importance of memory attacks increases. Persons who use encryption rely on their data to be protected against physical access, which hard disk encryption alone cannot provide. Main memory can no longer be regarded as a trusted resource because of cold boot and DMA attacks. Consequently, multiple counter measures have been developed to make disk encryption withstand memory attacks. One approach is to run the encryption algorithm only on the CPU without using memory [1,10,11]. Another solution are hard disks encrypting their data with a built-in crypto-module that stores keys securely in the disk itself. However, all these solutions have in common that they protect only the disk encryption key against main memory attacks. The memory contents of any program currently executed rests unprotected in RAM. An attacker can exploit this fact to obtain information about both, programs running on the target system, and the data they are operating on. Therefore, solutions are required to overcome the issue of sensitive data being openly accessible in RAM. Special software solutions already exist to protect private keys [12,13] during computations. However, these solutions are limited to computations with private keys. Hardware solutions such as Intel's software guard extensions (SGX) [14] could be used to protect RAM contents more generically. However, SGX has not been released by Intel. Hence, we provide a software only solution to protecting RAM contents during computations.

1.2 Contributions

To protect program code and data in RAM during computations, our contributions are as follows:

- We provide a Turing complete execution environment running on x86 commodity hardware, which allows program execution to treat RAM as untrusted. To this end, we use a bytecode interpreter executing programs without directly using RAM for code and data. This interpreter stores its state in CPU registers and uses RAM only to store encrypted data, effectively securing it against memory attacks.
- We provide a proof-of-concept implementation of the interpreter targeting the x86 architecture. It is delivered in form of a loadable kernel module compatible with recent Linux kernels. The interpreter can be used as the central part of a software-only trusted computing base.
- We evaluate the interpreter with regard to several attack types. Concerning memory attacks, we show that the interpreter fulfills our goals and is fully secure against those kind of attacks. Against attacks on the software level, the interpreter provides a considerable security add-on that can protect the confidentiality of executed programs even against attackers with root privileges.

– We benchmarked the interpreter against three other programming languages, namely C, Java, and Python. The results show that C and Java are both between one or two magnitudes faster than both Python and our interpreter, which is not surprising considering that these languages utilize native code execution. Between our interpreter and Python, the difference in performance is much smaller, with Python being faster than the interpreter by an average factor of 4.

1.3 Outline

In Sect. 2, the design and implementation of our interpreter is described. Subsect. 2.1 introduces different parts the interpreter consists of, and how they interact with each other. Subsect. 2.2 depicts where and how the interpreter manages the state of an executed program. In Subsect. 2.3, we discuss how the encryption algorithm of TRESOR [1] was adapted to fit our needs, and how encryption is applied to the interpreter data. Subsect. 2.4 shows the steps the interpreter goes through while executing a program. Our implementation is evaluated in regards to several aspects of performance and security in Sect. 3. In chapter Sect. 4 we review other solutions to protecting RAM contents during computation. Last, Sect. 5 contains a discussion about limitations and ideas for further developments.

2 Implementation

In the following we describe the design and implementation of our interpreter. While implementing the interpreter, we have to keep two security policies in mind. First, we are not allowed to use main memory for any sensitive data, as memory is considered untrusted. Second, we should not weaken the given security of the system provided by TRESOR [1]. We solve the first challenge by enforcing that any data is encrypted before it hits memory. The second task is fulfilled by ensuring the confidentiality of the TRESOR key during interpreter runtime.

2.1 General Interpreter Composition

In this section, we show what the different parts the interpreter consists of are, what their purpose is, and how they interact. We do this by walking through a program's life cycle from being programmed over compilation and execution to termination. The interpreter consists of three parts: the front-end, running in user-mode, which takes encrypted binary programs as input and outputs the results of the calculations, and the back-end, running in kernel mode, which does the actual interpretation of the given encrypted program. Additionally, a compiler tool is provided. It compiles programs from a simplified C dialect to interpreter bytecode, and encrypts them afterwards. A general overview of the layout is given by Fig. 1.

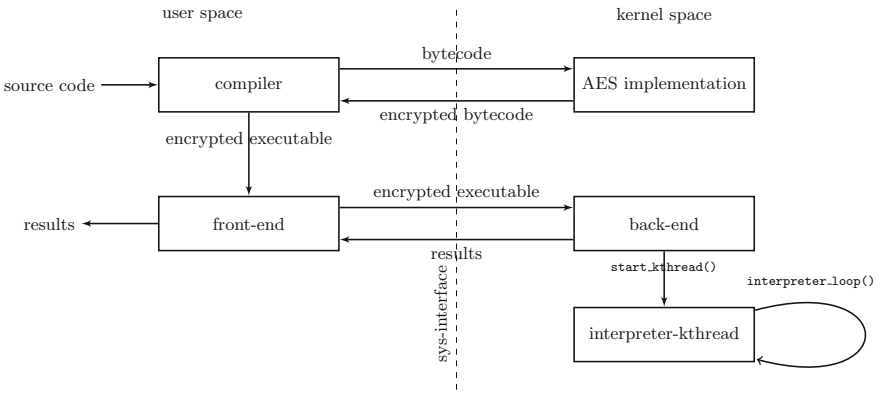

Fig. 1. The interpreter is separated into a compiler and front-end in user space, and a back-end with the AES implementation in kernel space. The different parts communicate over the kernel's sys-interface. On program execution, the back-end starts a kernel thread running the interpreter loop.

At first, the Linux kernel has to be booted up. At this point, TRESOR asks for a password which is used to derive the encryption key. During the system's life time, every program created will be encrypted with this key, and every program the interpreter executes will be decrypted with this key. After a password is entered, a program to be executed can be created. For this task, a simple programming language was devised to avoid having to program directly in bytecode. This programming language is called "*secure C-like language*". Its files are called `.scll`. It is based on a reduced subset of C that lacks features such as arrays and global variables. The grammar of SCLL is given in Appendix A.1.

A finished program is passed to the compiler to translate into bytecode. After compilation, the bytecode is not yet ready for execution; to meet our goal of secure execution, it has to be encrypted first. Encryption needs the key which is stored in debug registers by TRESOR. Therefore the interpreter back-end runs in kernel space and provides the necessary encryption facilities; these are made accessible to user space programs through the kernel sys-interface `/sys/kernel/bispe/crypto`. The compiler utilizes this and sends the unencrypted bytecode through the sys-interface to kernel space, where it gets encrypted with the currently set key. After getting back the encrypted bytecode, the program is outputted as an encrypted executable file, now with the extension `.scle` (for "*secure C-like executable*").

In order to execute our encrypted program, the interpreter front-end is used. The front-end acts as a user mode wrapper to the functionalities exported by the back-end. After its call, the front-end invokes interpretation of the program by passing the program to the back-end, again through the sys-interface. The sys-entry for this is `/sys/kernel/bispe/invoke`. Alongside the program, additional information is passed to the back-end, e.g. command line arguments and buffers providing space for execution results. The front-end then blocks until the back-end has finished execution of the program.

Before execution begins, the back-end first has to initialize the execution environment. Most notably, this means allocating the different memory segments the interpreter uses. These are the code segment, the operand stack, and the call stack. The just allocated code segment gets pre-filled with the encrypted program. The different segments and their usage are described in detail in Sect. 2.2. After initialization, the back-end creates a new kernel thread which runs the interpretation. There are two reasons to use a kernel thread instead of starting the interpretation directly in the back-end thread. First, it allows for clean signal handling. If the user gets impatient and stops the front-end before the execution is fully done, for instance with a `SIGINT` signal, the back-end must ensure that all kernel memory is freed before returning back to user space. With a kernel thread running the interpretation, the back-end thread just sleeps until execution is done, and if the sleep is interrupted by a signal, the kernel thread is issued to stop execution. The kernel thread notices that it should stop, and releases all allocated kernel memory. The second reason are future extensions: Currently, only one interpretation can be run at a time. If the interpretation is executed as a separate thread, it is easier to extend the program to allow multiple concurrent interpretations in the future.

To begin execution, the kernel thread repeats the interpreter loop. The interpreter loop is described in detail in Sect. 2.4. If the program is either finished, an error occurred, or the interpreter is ordered to cancel by the user, the loop is stopped. The kernel thread wakes up the sleeping back-end, reports the execution results, and finally terminates. Back in the main thread, the execution results get copied back to the front-end. Last, all allocated kernel resources are released, and control flow returns to user space. The front-end unblocks, reads out the execution results and potential output data is presented to the user.

2.2 Interpreter Memory Layout

This section details how the interpreter organizes the executed program's memory, and in which way the encryption interacts with the data. The interpreter is simulating a simple *stack machine*. This means that arithmetic and logical instructions always take their operands from top of a stack structure, on which they also put their computation results. The reason why a stack based interpreter is chosen over a register based one is that the bytecode instruction set is simplified, even though register based interpreters have been found to offer better performance [15].

The interpreter uses a unified word size of four byte for every instruction and every data element, which simplifies data accesses. An instruction consists of a four byte opcode and a four byte argument, if the instruction specifies one. The unit by which the interpreter accesses memory is per row consisting of 16 byte. Every time the interpreter reads from memory, it reads in a full row, even though the requested data is only of word size, because in memory, there is only *encrypted* data. The AES algorithm, by which this data is en- and decrypted, uses a block size of 16 byte. A program uses three memory segments during its execution: the code segment, the operand stack segment, and the call stack segment. Each segment's start address is aligned to 16 byte.

The *code segment* stores the code of the program. Before execution starts, the encrypted bytecode is relocated to this segment. The interpreters instruction pointer pointing into the code segment can be padded with random data before encryption so an adversary can not deduce program flow from it.

Intermediate data is stored in the *operand stack segment*, with a stack pointer pointing to the top of the stack. Every instruction that works on data expects its arguments and leaves its result on the operand stack. The only exceptions are load and store instructions which can transfer data between the operand stack and variables. Subroutines leave their result on this stack.

The *call stack segment* stores function related data. Every time a subroutine is called, a new stack frame is generated on top of the call stack. This frame contains subroutine arguments, local variables, and the return address, which is the address where execution is resumed after the subroutine ends. Like the operand stack, the call stack has a pointer pointing to its top.

We now take a look at how data from the segments can be used by the interpreter, although it rests encrypted in memory. To this end, throughout runtime, a decrypted 16 byte slice of each segment is held in a so-called *row register*, that is one of the 16 byte SSE registers. In case of the code and operand stack segments, this slice is always the row the instruction or stack pointer currently points to. For the call stack segment, it is the row which contains the data element currently processed. Instructions can now process their required data, because this data is always present in decrypted form.

When an element is requested from memory, the base address of the containing row is calculated from which 16 byte are copied from memory to the corresponding row register. The register gets decrypted and the data is almost ready to be accessed. Before a bytecode instruction can actually use it, the element needs to be extracted from the register row to a general purpose register.

Let us suppose an instruction has altered an element in some way and wants to push that element on top of the operand stack. The element now takes the inverse way back to memory. First, the stack pointer gets increased by four bytes to make space for the new element. Second, the element gets inserted in the stack row register at the four byte offset specified by the stack pointer, relative to the base address of the row. Further instructions are processed until eventually the stack pointer is increased so far that it exceeds the current row and points into a new row. To make space for the new stack row, the current row register must then be moved away to memory. This happens by encrypting the stack row register, and saving it to the corresponding memory location. New data elements can then be inserted in the empty stack row register. This procedure is largely the same for the code and the call stack segment.

2.3 Encryption Scheme

The interpreter utilizes the AES-256 encryption and decryption routines provided by TRESOR. TRESOR uses the SSE registers to store the AES round keys. However, this leaves no space for the interpreter state inside the SSE registers. Fortunately, with the introduction of the Advanced Vector Extensions

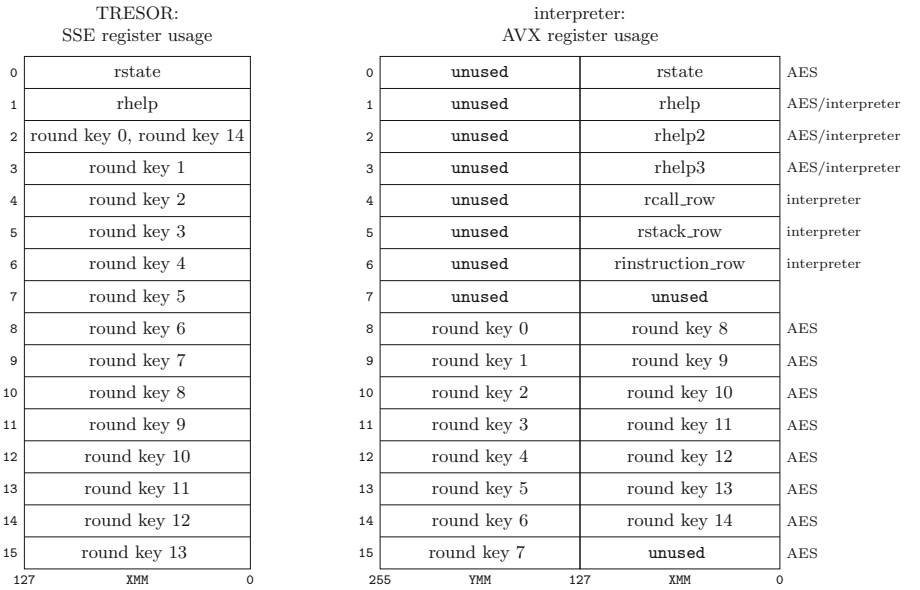

Fig. 2. Modification of TRESOR's register allocation to host the interpreter state.

(AVX), the size of the SSE registers was increased from 128 bits to 256 bits [16]. This allows us to place two round keys in each of the 256 bit registers, cutting the amount of registers needed for round keys in half. Figure 2 displays how the register distribution was changed from TRESOR to the interpreter, and also which registers are used for encryption, and which for the interpreter.

The interpreter uses *cipher block chaining* (CBC) [17] as its cipher mode. For both stack segments, the IV is created at runtime, directly after the segments were allocated, before execution start. The interpreter writes 128 random bits to the beginning of the segment, obtained from the kernel function `get_random_bytes`, which uses the Linux kernel's internal pseudo random number generator (PRNG). For the code segment, the IV is determined at compile time, from `/dev/urandom`, and it stays the same for the executable until a recompilation occurs.

The crypto-routines of the interpreter are in `backend/bispe_crypto_asm.S`. Programmatically, most of TRESOR's code could be carried over, but a few changes were made. Those changes mostly address access to the AVX registers instead of SSE for key material. Further, the crypto module of the interpreter was extended with routines to encrypt memory in CBC mode (`bispe_encblk_mem_cbc`), as well as encrypting a register in place (`bispe_encblk`).

The implementation of the CBC mode for memory segments during runtime is, in all but one case, trivial. The non-trivial case occurs because the call stack segment allows writing to arbitrary elements; a saving of the call stack row register to memory triggers a re-encryption from the changed block up to the

end of the segment. As writes to the stack segment only target the frame of the current function, however, the chain to be encrypted is short.

2.4 The Interpreter Loop

When executing a program, the interpreter has to repeat the same set of steps for every instruction. The instruction specified by the instruction pointer is fetched from memory and the interpreter performs the appropriate actions to execute the instructions, based on the fetched opcode. Afterwards, the instruction pointer is changed to point to the next instruction. These steps basically get repeated in a loop until the program is finished.

Figure 3 lists the individual steps taken during the interpreter loop in a textual manner. Figure 4 shows a flowchart version of the process. The interpreter splits up this loop in individual cycles – which are not to be confused with the above mentioned fetch-execute cycle.

1. Begin atomic CPU section by disabling scheduling and interrupts.
2. Generate AES round keys in AVX registers.
3. Load program state to registers.
4. Repeat until cycle has ended or halt flag is set:
   4.1. Extract opcode of current instruction from instruction row.
   4.2. Jump to instruction routine specified by opcode.
   4.3. Execute instruction routine.
   4.4. Increase instruction pointer. If new instruction is not present in current instruction row, load and decrypt next instruction row from memory.
5. Save program state encrypted to memory, clear AVX registers.
6. End atomic CPU section by enabling interrupts and scheduling.
7. If halt flag is set or execution was stopped by the user, break interpreter loop. Otherwise, go back to step 1.

Fig. 3. Individual steps the interpreter repeats during a cycle.

A cycle consists of multiple instructions executed subsequently. The amount of instructions executed in one cycle is not fixed but user configurable. From the outside, a cycle represents an atomic unit. Therefore, one cycle always runs uninterruptible and even the kernel is not able to interrupt execution. This is necessary to protect the integrity of both the program and the cipher key, as during the cycle, the AVX registers hold decrypted program- and encryption state, and these registers are principally free to access for any process.

Since the interpreter runs in kernel mode, it has access to kernel functions which can create an atomic section. Preemptive kernel scheduling can be disabled with the function `preempt_disable`. To provide true atomicity, `local_irq_save` has to be called for disabling interrupts.

At the beginning of a cycle, after an atomic section has been started, the interpreter first derives the round keys from the cipher key and places them in the round key registers. The next step is to load the program state into registers, so that the interpreter instructions can operate properly. This means that

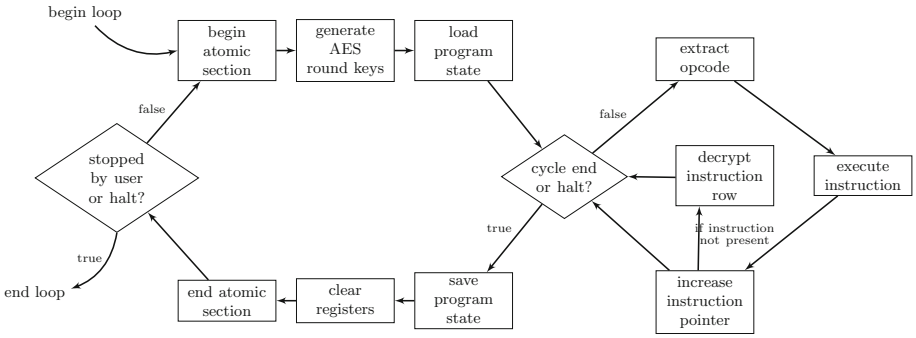

Fig. 4. The different steps the interpreter repeats in the interpreter loop. One pass represents an atomic cycle, in which program data and encryption state are protected from leaking to memory.

for each segment, a row gets decrypted to the corresponding row register, as described in Sect. 2.2. Internally, every segment pointer in memory is mapped to a general purpose CPU register, e.g. the instruction pointer to register `r11`, because these pointers have to be accessed often during program execution. In the "load program state step", the state pointers get loaded from memory in their respective register. Then the interpreter is ready to process bytecode instructions. The used technique is *indirect threading*; each bytecode instruction is represented by a single routine in `backend/asm_instructions.S`. When one of these routines gets executed, it simulates the bytecode instruction it represents on the programs state. The addresses of each of those routines are stored in a jump table. To execute a bytecode instruction, the interpreter extracts the opcode of the current instruction from the instruction row and calls the appropriate routine to process the instruction. In Appendix A.3, the functionality of each bytecode instruction is explained.

After the instruction is finished, the instruction pointer gets updated to point to the instruction to be executed next. Before a new instruction is executed, however, it is checked if the amount of instructions executed in this cycle exceed the specified maximum amount of instructions in one cycle or the halt flag was set, e.g. by the special `finish` instruction, or due to the occurrence of a runtime error, e.g. a division through zero, or a stack overflow. If either of this is the case, the next instruction is not executed. Instead, the interpreter saves the current program state to memory. This is done by encrypting the row registers and saving them to memory. Additionally, the values of the state pointer registers are written to their counterpart in memory again. Before the atomic section is left again, it is important to reset the content of the AVX registers, before anyone else can have access to them. At last, the atomic section is ended by activating scheduling and interrupts again.

As the interpreter code itself is only executed during an atomic section, an adversary has no way to observe and infer any knowledge from the `rip` instruction pointer pointing to interpreter code execution.

However, when a subroutine within the interpreter code is called, the *return address* is pushed on the stack. The leaked information about a single instruction causes nearly no actual damage, but it is desirable to thwart even theoretical attacks. The solution is to not use the `call` instruction directly but rather store the current position in a register (`%r9`) and jump into the function. For returning, instead of calling `ret` an indirect jump to an address stored in a register (`jmp *%r9`) is used.

3 Evaluation

We evaluate our interpreter concept and its implementation with respect to two criteria. In Subsect. 3.1, we discuss benchmarks comparing the performance of the interpreter against other programming languages. And in Subsect. 3.2, we deliver an analysis of the interpreter's security against memory attacks, software attacks, and hardware attacks.

3.1 Performance

In this section, we investigate the performance of our interpreter. Given the interpreter's design, a performance drop-off compared to other execution environments must be expected due to encryption. The interesting question is how big the difference in performance compared to other programming languages is. To test this, benchmarks with four different language environments were performed, one of them being our own bytecode interpreter. All benchmarks were performed on an Intel Core i5-3570K CPU which supports AVX as well as AES-NI. The operating system is Xubuntu Linux 14.04 with kernel version 3.8.2 and TRESOR patch applied.

The other three languages (C, Java and Python) were selected to fit into different kinds of execution types. As C compiles to machine code which can be executed natively by the CPU, C should run the fastest among the tested languages. The C programs have been compiled with GCC version 4.8.2. The second language chosen is the Java language. Java represents the class of JIT-compiled interpreted languages. Therefore, we expected Java to perform quite well albeit a bit slower than C. The Java version used was OpenJDK 1.7.0_65. The third and last language choice is the Python language, using the default Python implementation in version 2.7.6. Python represents a simpler kind of interpreter implementation. It uses no JIT compilation, and the source code is parsed to bytecode just before execution. This makes Python a slower type of interpreter, which makes its performance results closest to our interpreter's performance.

We benchmarked the following mathematical algorithms:

- the nth *Fibonacci numbers*
- the first n *Prime numbers*
- the *Pascal triangle* with n rows.

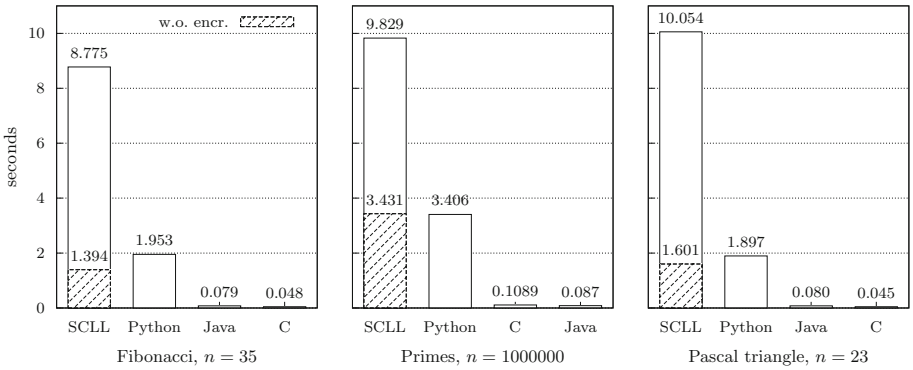

Fig. 5. The results of the language benchmark show that the interpreter is vastly slower than C and Java, but still within performance reach of Python.

The implementation of the three algorithms in the interpreter's language. It can be found in Appendix A.2. The elapsed time of a program run has been measured with the built-in `time` shell command of the Bash shell.

An additional property of the interpreter is that execution is divided in atomic cycles. Each of these cycles creates a performance overhead. We tested the exact influence of this setting on performance and have chosen the default instructions per cycle value of the interpreter to be 2000. This was also the value used for performance tests.

The benchmark results are listed in Fig. 5, with the results averaged out from 50 program runs. The column labeled "SCLL" contains the benchmark results of our interpreter. A first overview shows roughly the same picture for all three programs. As expected, the interpreter is slower than Python, and C and Java perform both much faster than Python and SCLL. C and Java are between one or two magnitudes faster – their bars in the figure are only barely visible. This is owed to the power of native code execution, for Java enabled through JIT. If we compare our interpreter to Python, we see that the interpreter performs reasonably well. On average, SCLL is about a factor 4 slower than Python.

The influence of encryption on the interpreter's performance is interesting, and we decided to measure it explicitly. In Fig. 5, the results of this benchmark are shown as striped bars within the SCLL bar. It is easy to see that encryption takes up a major part of the interpreter's runtime. On average, the interpreter spends four fifth of the overall runtime with encryption.

If we contrast the individual programs, we can see that in respect to the other languages, the interpreter performs best on the Primes program. Whereas performance for Fibonacci and Pascal is similar, the ratio to the other languages is the best for Primes. Looking for the reasons, we have to take a look at the source codes of the programs (see Appendix A.2). We can see that Primes is implemented purely iterative, Fibonacci recursive, and Pascal uses both iteration and recursion. This indicates that recursive programs affect the interpreter's performance negatively. Recursion requires the repetitive use of function calls.

Within each function call, a stack frame has to be allocated and freed which creates encryption overhead at the interpreter. In the Primes program, encryption takes up two thirds of the running time, whereas for Fibonacci and Pascal, the overall execution time is composed of five sixths of encryption time.

Summarizing, we have benchmarked the interpreter against three other popular programming languages: C, Java, and Python. The interpreter is slowed down considerably by the overhead caused by encryption, and without it, the performance of the implementation is on par with Python.

3.2 Security

The most important property of our interpreter is the security it can provide. In Sect. 3.2, we show that the interpreter holds its designated goal and is secure against attacks on memory such as cold boot attacks. We also discuss how far the interpreter is protected against software attacks in Sect. 3.2 and investigate possible weaknesses of the interpreter against hardware attacks in Sect. 3.2.

Protection Against Memory Attacks. As outlined in Sect. 2, great care has been taken to ensure that no sensitive interpreter state or even encryption keys are leaked into RAM. However, we now like to practically uphold this fact. To this end, we perform memory scans of a system running the interpreter. We used a Qemu/KVM virtual machine running Debian Linux with kernel version 3.8.2 (TRESOR patched) to obtain memory images.

Three memory dumps were taken at different times. The first without running the interpreter at all, to compare if running the interpreter influenced the scan results. Afterwards, the interpreter kernel module was loaded, and the interpreter ran the program calculating the Fibonacci numbers from the previous section. The second memory dump was taken during the interpreter running the program, and a third one after the Fibonacci program was executed a hundred times. We searched the memory dump for both AES key schedule patterns as well as cleartext patterns of bytecode programs that are decrypted by the interpreter. Only small matches could be found that can be attributed to coincidence, because searching for any random pattern also yields matches of the same length.

These results indicate that neither debug registers nor AVX registers get leaked to memory, which confirms the adherence of the interpreter's security goal to leak no sensitive data in memory. In sum, we can state that the interpreter's implementation protects executed programs against memory attacks.

Protection Against Software Attacks. Another interesting topic is the level of protection the interpreter can provide for executed programs against attacks on the software level. That is, the attacker model now switches from physical access to logical access to the system the interpreter is running on.

One idea is to pick off the data during interpreter runtime when the interpreter is processing it in decrypted form. The interpreter holds decrypted data in the segment row registers. If an attacker can continuously copy the content

of those row registers, while the interpreter is running the program, a complete picture of the program's code can be reconstructed, as well as the data the program is working with. This would, however, require outside access to the row registers at the time they contain decrypted data. Since the row registers hold decrypted data only within the atomic CPU section, this is not easily possible. The atomic section prevents any other process that could read out the registers from running on that CPU, and the atomic section can only be ended by the interpreter itself – even the kernel can not interrupt it. Attackers with system privileges can change the code of our kernel module such that no atomic section is entered before interpretation.

Attackers with system privileges, however, also have another attack surface. The encryption key is stored in the CPU debug registers at all times. Debug registers are a privileged resource which means that ring 0, the kernel, can access them. The debug registers are accessible for user space applications only through the `ptrace` system call. TRESOR patches certain kernel code to make the debug register inaccessible – the `ptrace` system call is patched to not return the debug register content.

It is impossible for a non-privileged attacker to read out the debug registers. Only an attacker with root privileges has more possibilities. By using a loadable kernel module (LKM), or `/dev/kmem`, arbitrary code can be inserted into the kernel and executed from within ring 0. In its current form, the interpreter is vulnerable to attacks of this kind. To protect against this security flaw, the TRESOR authors advise to compile the kernel without support for LKM and `/dev/kmem`, as then even root attackers are unable to access the cipher key. For now, the interpreter is designed as a LKM itself, but it would be possible to change the module into a kernel patch, which would allow hard compiling the interpreter into the kernel, while disabling support for LKMs. An attacker would then be required to use some kind of kernel exploit that allows to execute code. All in all, this would make the interpreter also resistant against most attacks on the software level.

Protection Against Hardware Attacks. A simple hardware attack would succeed if CPU registers keep their content after rebooting of the system. Fortunately, according to the authors of TRESOR, this is not the case.

In our discussion of the interpreter's resistance against attacks on memory, we mostly had the cold boot scenario in mind. DMA attacks, however, are also viable. Blass and Robertson [18] introduce an attack on CPU based encryption systems, exploiting DMA to write malicious code to kernel memory. In fact, the attack is named "TRESOR-Hunt" explicitly targeting TRESOR. Through patching the interrupt descriptor table, the kernel is issued to execute a payload, which is essentially a piece of code that dumps the debug registers to memory. After the cipher key is in memory, the attacker can obtain it via DMA. This attack, however, can be defended by device whitelisting, to only allow known devices to use DMA, or using a input/output memory management unit (IOMMU) to block critical memory regions from DMA access.

Physical access to the CPU also enables other kinds of attacks. The JTAG interface of a microprocessor allows an engineer to debug the running processor by connecting with the JTAG ports on the physical device. Some modern Intel CPUs also expose JTAG ports on their surface, which can be used to read out the debug registers. However, we are not aware of anyone successfully carrying out such an attack.

4 Related Work

Different solutions for CPU bound encryption on x86 exist such as TRESOR [1] and LoopAmnesia [11]. Also for ARM, a cold boot resistant implementation named ARMORED [19] has been developed. However, CPU bound encryption alone can only protect single encryption keys, which are mostly used as disk encryption keys as explained above. In contrast, we want to protect entire programs and their data against memory attacks. One solution which comes into mind for supporting the execution of encrypted programs is *Frozen cache* [10]. Frozen cache, however, must reserve the entire CPU cache which renders this approach unfeasible as it slows down the overall system performance too much.

In the past, encrypted program execution has already been worked at in different ways: Brenner et al. [20] have shown that secure program execution would be possible with a fully homomorphic encryption scheme [21]. They focus on securing programs in an untrusted environment, e.g. in cloud computing, which is not the primary goal of this work. Another approach is to use full memory encryption [22,23], which would indeed protect programs against memory attacks. However, software-based full memory encryption suffers from considerable performance drawbacks, while hardware-based full memory encryption is not available for end-users. Duc and Keryell [22], for example, rely on their own, special hardware architecture. [23] is restricted to ARM processors equipped with security hardware, while [22] relies on its own, special hardware architecture. Another memory encryption solution explicitly designed to mitigate cold boot attacks is Cryptkeeper [24]. Unfortunately, on its own, Cryptkeeper poses no viable solution because the encryption key itself is stored in clear in RAM.

Working special solutions are PRIME [12] and Copker [13]. However, they are restricted to computations with private keys.

A possible future technology to solve the issues surrounding secure program execution is Intel's Software Guard Extensions (SGX) [14]. SGX allows applications to run in so-called *enclaves*, which are secure memory containers inaccessible by anyone but the application itself. To achieve this, enclave memory is encrypted in hardware, with the encryption key stored securely in hardware. The system is explicitly designed to both protect programs against memory attacks and to enable running them securely in an untrusted environment. While being announced by Intel in 2013, it is still unknown when the first CPUs supporting SGX are released to the public.

Last but not least, this work was partially inspired by Breuer and Bowen [25]. They propose a "crypto-processor unit" (KPU) which instructions and data enter

and leave encrypted only. While the concept is not practically feasible yet, the underlying idea was useful to us in providing a generic software only protection mechanism for code and data during computations.

5 Conclusion and Future Work

In this chapter, we draw conclusions about our developed bytecode interpreter for secure program execution. In the previous chapters, we evaluated the interpreter regarding performance and security. We summarize the found limitations in Sect. 5.1. In Sect. 5.2, we present future tasks and investigations that can be pursued to further extend the interpreter's scope of applicability. Finally, in Sect. 5.3, we summarize the work done in this thesis and draw a conclusion about the overall applicability of our interpreter concept.

5.1 Limitations

Currently, the developed bytecode language supports only a narrow set of features. However, we have not yet found any obstacles caused by the interpreter's design which will impede future integration of common programming language features.

5.2 Future Work

In this section, we present some possible future developments. A certainly worthwhile goal is to extend the bytecode language to be fully compatible with the C language. This would make encrypted program execution through the interpreter widely applicable, as many programs are written in C, and several other programming languages can be compiled to C. The longterm goal is to be able to securely run everyday software, like text editors, browsers, and mail-, or office programs through the interpreter.

Usability must also be increased. Currently, a program can be executed and encrypted only with a single key; and for changing the key, the system has to be rebooted, which is quite inconvenient. It would be desirable, that the user is able to specify a password to use for encryption at compilation, and to enter that password again for execution. To implement this, the user password must be scrambled with the master key set by TRESOR.

Advances in performance can be gained by Intel's AVX-512 instruction set. AVX-512 increases the amount of SIMD registers to 32, and the register width to 512 bits, which yields four times more available register space than AVX has. The interpreter can use the additional space for caching. AVX-512 also brings many new assembler instructions, which may allow simplifying the code of the current implementation, yielding performance gains as well.

There are further cipher modes that also guarantee authenticity of the encrypted data, but that can be more complex to implement, so the interpreter currently limits itself to CBC. If it turns out that other cipher modes are needed, they can be integrated in the future.

5.3 Conclusion

Physical security has always been a weak point in the defenses of computer systems, especially mobile systems. Regardless of software protection measures, the data of a stolen laptop can easily be obtained by reading out the hard disk. As full disk encryption became common and closed a simple attack vector, attacks moved a level lower, targeting the disk encryption keys within the unencrypted RAM instead. Several kind of attacks on memory have been shown viable. Executing programs outside memory, using memory only for encrypted data, would protect sensitive user data against memory attacks.

In this work, we have shown how encrypted program execution is feasible when treating main memory as untrusted. The design consists of an interpreter which executes encrypted bytecode programs without using RAM for sensitive data. The program's bytecode and data is held decrypted only within CPU registers that are processed by the interpreter.

We provide a working proof-of-concept implementation for the x86 architecture, in form of a kernel module for the Linux kernel. Our proof-of-concept implementation supports a simple bytecode language, but we have shown that the concept can be extended to include more features soon.

To analyze the interpreter's resistance against memory attacks, several memory dumps were taken and scanned for patterns of encryption keys, round keys as well as code and data of executed programs. A significantly long byte pattern, that would indicate the interpreter leaking to memory, could not be found in any case. Furthermore, this result is strengthened by the fact that the interpreter uses the same protection approach as TRESOR which was thoroughly controlled before. The interpreter resists attacks on the software level as long as the attacker has no ring 0 privileges, as then the debug registers are no longer a secure storage for cryptographic keys. Without further measures, the interpreter is vulnerable to a special DMA attack [18], which inserts malicious code into the kernel to obtain keys. This attack can be mitigated by restricting DMA, as supported by recent Linux kernels.

Acknowledgement. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

A Appendix

A.1 SCLL Grammar

Listing 1.1. Grammar of SCLL in Extended Backus-Naur Form.

```
integer       =  digit, {digit} ;
identifier    =  (letter | '_'), { (letter | digit | '_') } ;
type          =  'void' | 'int' ;
num_op        =  '+' | '-' | '*' | '/' | '%';
bool_op       =  '==' | '!=' | '>' | '<' | '<=' | '>=' ;
fcall_arglist =  [expression, ',', [fcall_arglist]] ;
fcall         =  identifier, '(', fcall_arg_arglist, ')' ;
```

```
expression     =  (fcall
                     | identifier
                     | ['-'], integer
                     | '(', expression, ')'), [num_op, expression] ;
var_def        =  type, identifier, ['=', expression] ;
var_assign     =  identifier '=' expression ;
print          =  'print', expression;
return         =  'return', [expression] ;
cond           =  expression, bool_op, expression ;
branch         =  'if', '(', cond, ')', '{', sequence, '}',
                     ['else', '{', sequence, '}'] ;
loop_head      =  (var_def | var_assign), cond, var_assign ;
loop           =  'while', '(', cond, ')', '{', sequence, '}'
                     | 'for', '(', loop_head, ')', '{', sequence, '}'
                     | 'do', '{', sequence, '}', 'while', '(', cond, ')', ';' ;
statement      =  (var_def | var_assign | fcall | print | return) ';' ;
sequence       =  (statement | branch | loop), [sequence] ;
argdefnce      =  type | identifier ;
arglist        =  argdef, [',', arglist] | 'void' ;
func           =  type, identifier, '(', arglist, ')', ('{', sequence, '}' | ';') ;
program        =  [func, program] ;
```

A.2 Source Codes

In this section, the SCLL source codes of programs that were used to evaluate our work are listed. In particular, the programs 1.2, 1.3, and 1.4 were used in benchmarking (Fig. 6).

Listing 1.2. Program calculating the `n`'th Fibonacci number.

```
int fib(int i);

int fib(int i) {
        if(i == 1) return 1;
        if(i == 2) return 1;
        return fib(i-1) + fib(i-2);
}

void main(int n) {
        print fib(n);
}
```

Listing 1.3. Program calculating the primes to `primes`.

```
void print_prime(int p) {
        if(p % 2 == 0) return;

        for(int i = 3; i * i <= p; i = i + 2) {
                if(p % i == 0) return;
        }
        print p;
}

void main(int primes) {
        for(int i = 2; i <= primes; i = i + 1)
                print_prime(i);
}
```

Listing 1.4. Program calculating the pascal triangle with `max_row` rows.

```
int binom(int n, int k);

int binom(int n, int k) {
    if(k == 0) return 1;
    if(n == k) return 1;
    return binom(n-1, k-1) + binom(n-1, k);
}

void main(int max_row) {
    for(int n = 0; n < max_row; n = n + 1) {
            for(int k = 0; k < n+1; k = k + 1)
                    print binom(n, k);
    }
}
```

A.3 Bytecode Language

opcode	mnemonic	argument	operand stack: before → after	description
0	`nop`			perform no operation
1	`finish`			halt execution
2	`push`	value	→ value	push integer *value* on the stack
3	`print`		value →	write *value* to output buffer
4	`load`	index	→ value	load *value* from local variable at *index*
5	`store`	index	value →	save *value* to local variable at *index*
6	`add`		value1, value2 → result	add two integers, `r = v2 + v1`
7	`sub`		value1, value2 → result	subtract two integers, `r = v2 - v1`
8	`mul`		value1, value2 → result	multiply two integers, `r = v2 * v1`
9	`div`		value1, value2 → result	divide two integers, `r = v2 / v1`
10	`mod`		value1, value2 → result	remainder of two integers, `r = v2 % v1`
11	`jmp`	address		jump to instruction at *address*
12	`jeq`	address	value1, value2 →	if *value2* is equal to *value1*, jump to instruction at *address*
13	`jne`	address	value1, value2 →	if *value2* is not equal to *value1*, jump to instruction at *address*
14	`jl`	address	value1, value2 →	if *value2* is less than *value1*, jump to instruction at *address*
15	`jle`	address	value1, value2 →	if *value2* is less than or equal to *value1*, jump to instruction at *address*
16	`jg`	address	value1, value2 →	if *value2* is greater than *value1*, jump to instruction at *address*
17	`jge`	address	value1, value2 →	if *value2* is greater or equal to *value1*, jump to instruction at *address*
18	`call`	address		call subroutine at *address*
19	`ret`			return from subroutine
20	`prolog`	amount		allocate space for *amount* elements on the call stack (subroutine prolog)
21	`epilog`	amount		free space of *amount* elements on the call stack (subroutine epilog)
22	`argload`	index		save command line argument *index* to local variable at *index*

Fig. 6. Complete listing of all bytecode instructions.

References

1. Müller, T., Freiling, F.C., Dewald, A.: Tresor runs encryption securely outside ram. In: Proceedings of the 20th USENIX Conference on Security (SEC 2011), pp. 17–17. USENIX Association, Berkeley (2011)
2. Alex Halderman, J., Schoen, S.D., Clarkson, W., Heninger, N., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009). doi:10.1145/1506409.1506429
3. Gruhn, M., Müller, T.: On the practicability of cold boot attacks. In IEEE Conference Publications, editor, Eighth International Conference on Availability, Reliability and Security (ARES), pp. 390–397 (2013)
4. A Guide to Understanding Data Remanence in Automated Information Systems. NCSC-TG-025, National Computer Security Centre, Sep 1991
5. Gutmann, P.: Data remanence in semiconductor devices. In: Proceedings of the 10th Conference on USENIX Security Symposium, SSYM 2001, vol. 10. USENIX Association, Berkeley (2001)
6. Skorobogatov, S.: Low temperature data remanence in static RAM. Technical report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, Jun 2002

7. Wyns, P., Anderson, R.L.: Low-temperature operation of silicon dynamic random-access memories. IEEE Trans. Electron. Devices **36**(8), 1423–1428 (1989). doi:10.1109/16.30954, ISSN 0018–9383
8. Becher, M., Dornseif, M., Klein, C.N.: FireWire: all your memory are belong to us. In: Proceedings of CanSecWest Applied Security Conference, Vancouver, British Columbia, Canada (2005)
9. Carrier, B.D., Grand, J.: A hardware-based memory acquisition procedure for digital investigations. Digital Invest. **1**(1), 50–60 (2004)
10. Pabel, J.: Frozen cache, Jan 2009. http://frozenchache.blogspot.com
11. Simmons, P.: Security through amnesia: a software-based solution to the cold boot attack on disk encryption. In: Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC 2011, pp. 73–82. ACM, New York (2011). ISBN 978-1-4503-0672-0
12. Garmany, B., Müller, T.: PRIME: private RSA infrastructure for memory-less encryption (best paper award). In: Applied Computer Security Associates (ACSA) and ACM (eds.) Proceedings of the 29th Annual Computer Security Applications Conference (2013)
13. Guan, L., Lin, J., Luo, B., Jing, J.: Copker: Computing with private keys without ram. In: Network and Distributed System Security Symposium (NDSS) (2014)
14. McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2013, pp. 10:1–10:1. ACM, New York (2013). doi:10.1145/2487726.2488368, ISBN 978-1-4503-2118-1
15. Shi, Y., Casey, K., Anton Ertl, M., Gregg, D.: Virtual machine showdown: stack versus registers. ACM Trans. Archit. Code Optim. 4(4), 2:1–2:36 (2008). doi:10.1145/1328195.1328197, ISSN 1544–3566
16. Lomont, C.: Introduction to Intel Advanced Vector Extensions. Intel Corporation, Jun 2011
17. National Institute for Standards and Technology. Recommendation for Block Cipher Modes of Operation, NIST Special Publication 800–38A edition, Dec 2001
18. Blass, E.-O., Robertson, W.: TRESOR-HUNT: attacking CPU-bound encryption. In: Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC 2012, pp. 71–78. ACM, New York (2012). doi:10.1145/2420950.2420961, ISBN 978-1-4503-1312-4
19. Götzfried, J., Müller, T.: ARMORED: CPU-bound encryption for android-driven ARM devices. In: Proceedings of the 2013 International Conference on Availability, Reliability and Security, ARES 2013, pp. 161–168. IEEE Computer Society, Washington, DC (2013). doi:10.1109/ARES.2013.23, ISBN 978-0-7695-5008-4
20. Brenner, M., Wiebelitz, J., von Voigt, G., Smith, M.: Secret program execution in the cloud applying homomorphic encryption. In: 2011 Proceedings of the 5th IEEE International Conference on Digital Ecosystems and Technologies Conference (DEST), pp. 114–119, May 2011. doi:10.1109/DEST.2011.5936608
21. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009). http://crypto.stanford.edu/craig
22. Duc, G., Keryell, R.: CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In: 22nd Annual Computer Security Applications Conference, ACSAC 2006, pp. 483–492, Dec 2006. doi:10.1109/ACSAC.2006.21

23. Henson, M., Taylor, S.: Beyond full disk encryption: protection on security-enhanced commodity processors. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 307–321. Springer, Heidelberg (2013)
24. Peterson, P.A.H.: Cryptkeeper: Improving security with encrypted RAM. In: 2010 IEEE International Conference on Technologies for Homeland Security (HST), pp. 120–126, Nov 2010. doi:10.1109/THS.2010.5655081
25. Breuer, P.T., Bowen, J.P.: A fully homomorphic crypto-processor design: correctness of a secret computer. In: Jürjens, J., Livshits, B., Scandariato, R. (eds.) ESSoS 2013. LNCS, vol. 7781, pp. 123–138. Springer, Heidelberg (2013)