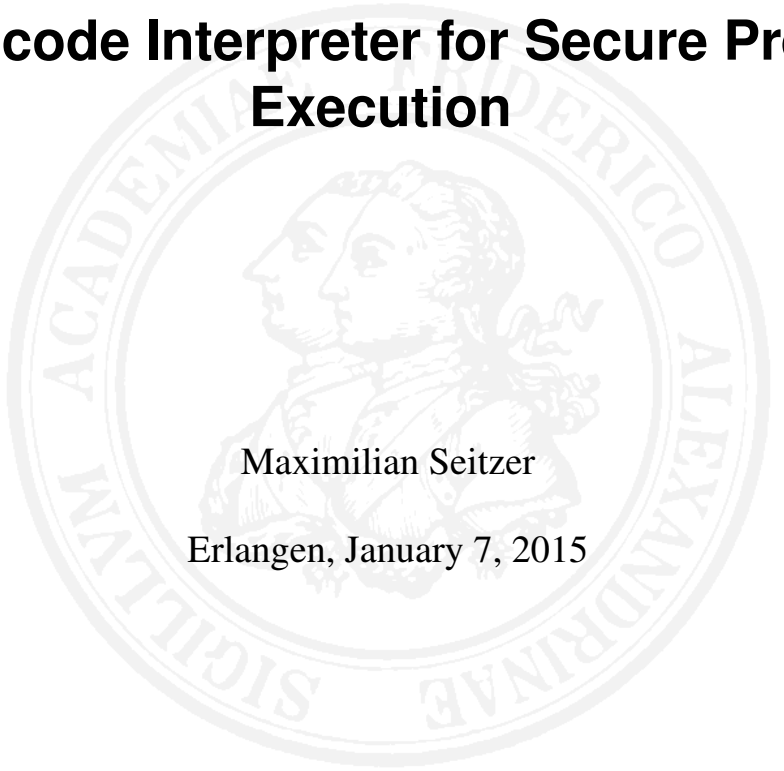
Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

BACHELOR THESIS

A Bytecode Interpreter for Secure Program Execution



Maximilian Seitzer

Erlangen, January 7, 2015

Examiner: Prof. Dr. Felix Freiling
Advisors: Dr. Tilo Müller
Michael Gruhn

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, January 7, 2015

[Maximilian Seitzer]

Zusammenfassung

Physischer Zugriff auf ein System ermöglicht einem Angreifer, den Arbeitsspeicher durch Kaltstart- und DMA Angriffe auszulesen. Bis jetzt schützen Gegenmaßnahmen nur gegen Angriffe auf Festplattenverschlüsselungsschlüssel, wobei der restliche Speicherinhalt anfällig bleibt. Diese Arbeit stellt einen Bytecode Interpreter vor, der Code und Daten von Programmen vor Speicherangriffen schützt, indem er sie ausführt, ohne den Arbeitsspeicher für vertrauliche Inhalte zu benutzen. Alle Programminhalte innerhalb des Speichers sind sicher verschlüsselt, wofür der Interpreter TRESOR einsetzt, eine Implementierung der AES Verschlüsselung, die resistent gegen Kaltstartattacken ist. Die Implementierung wurde als ein Linux Kernel Modul entwickelt, wobei Nutzen aus den CPU Befehlssätzen AVX für zusätzliche Register, und AES-NI für performante Verschlüsselung gezogen wurde. Die Arbeit zeigt, dass der Interpreter sicher gegen Speicherangriffe ist, und dass die Performanz trotz des Mehraufwands durch Verschlüsselung insgesamt akzeptabel ist.

Abstract

Physical access to a system allows an attacker 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. This thesis presents 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 safely encrypted, for which the interpreter utilizes TRESOR, a cold boot resistant implementation of the AES cipher. The implementation was developed as a Linux kernel module, taking advantage of the CPU instruction sets AVX for additional registers, and AES-NI for performant encryption. The thesis shows that the interpreter is secure against memory attacks, and that the overall performance is acceptable despite the penalty induced by encryption.

CONTENTS

1 INTRODUCTION

Physical security has often been the weak point in the defense of computer systems, especially of mobile devices. Against someone who has physical access to the device, software protection methods, e.g. account authentication, are no longer effective. Confidential data can be as simply read out as attaching the target's hard disk to an attacker controlled system. Nowadays, this trivial attack is usually no longer possible, as users began to utilize full disk encryption. Thus, attacks moved to a lower level, targeting RAM instead of the hard disk. As it stands, encryption is not applied to RAM, which makes memory attacks feasible. A memory attack is a physical attack that lets an adversary obtain a complete or partial memory image of the targeted running system. Such a memory image can be exploited for multiple purposes, as detailed later.

One such type of attack on memory is known as the "cold boot attack". This attack exploits the fact that the content of RAM chips is not instantly erased after power is disconnected from it. Instead, the chip's data gradually fades away and can be accessed for a short period of time after powering off [1]. This effect is known as data remanence [2]. By external cooling of the chip, this period can be prolonged to a few minutes, or even hours [3–5].

A cold boot attack typically works by rebooting a running system, and booting into a previously prepared tool, e.g. from an USB drive. The tool can then copy the content of the memory modules to the USB drive, yielding a complete memory image. To prolong the time the attacker has to initiate the copying process, the memory modules of the target system are cooled down to up to -50°C. For this, no special equipment is necessary, as off-the-shelf spray cans containing certain kinds of gas are sufficient. If the target prevents booting from USB, e.g. by setting a BIOS password, the attack is still executable by moving the memory modules from the target into a machine controlled by the attacker. The practicability of this attack on today's commonly used DRAM chips was shown by Halderman et al. [5] in 2009, which suddenly made the attack broadly known. Today, cold boot attacks are a recognized attack vector, and their feasibility was further validated, e.g. by Gruhn and Müller [6]. Cold boot attacks have even been shown to be possible against Android mobile phones [7].

Cold boot attacks are not the only kind of attack which allows to read out memory. Another threat are DMA attacks. DMA attacks exploit the fact that the direct memory access protocol allows external devices to directly interface with RAM, without the operating system being involved. The actual purpose of that is to allow data transfer between the external device and RAM at the highest speed possible, but this also permits the device to read out supposedly confidential data, like encryption keys. To conduct the attack, it is only necessary to connect a manipulated DMA device to a running system. This was shown in practice, e.g. for devices connected through Firewire ports [8, 9].

As the spread of full disk encryption extends, and devices become more and more mobile, the importance of memory attacks increases. This is especially true in the business sector, in which employees travel a lot, which opens many opportunity for their business laptop to get stolen, e.g. at an airport or train station. Persons who use encryption rely on their data to be protected by it, which hard disk encryption alone can currently not do.

1.1. Motivation

The main focus of the study by Halderman et al. [5] was to find hard disk encryption keys in the extracted memory images, to circumvent disk encryption. Consequently, multiple counter measures have been developed to make disk encryption withstand memory attacks. One approach is to make the encryption algorithm run without using memory, thus only on the CPU [10–13]. This is a widely accepted solution to mitigate memory attacks, and we will return to them later. Another solution are hard disks encrypting their data with a crypto-module, which stores the used key securely in hardware.

These solutions all have in common that they only protect encryption keys against memory attacks. However, an attacker may have other targets than that. The memory space of any program currently executed on a system rests openly in RAM. Even a terminated program leaves its memory remnants behind. Eventually, the locations of memory where these remnants lie is overwritten by other programs, but until then, the application data remains in RAM. An attacker can exploit this fact to obtain extensive information about what programs run on the target system were doing at the time of the memory attack. Among the sensitive data that can be leaked this way are passwords, private keys, authentication credentials, and banking data. And this is only sensitive data that the average user has. If the scope is extended to include business users, one can imagine finding sensitive employee data, trade secrets or prototypes on a stolen enterprise laptop. This scenario may sound unlikely to occur, but it is easily imaginable that actors involved in corporate espionage will spent considerable efforts to obtain the desired information from their target.

Therefore, a solution is required to overcome the issue of sensitive application data lying around openly accessible in RAM. An execution environment designed for programs to be executed in a secure way, i.e. without using RAM for data, would help to defeat memory attacks. In this environment, memory can no longer be regarded as a trusted resource.

1.2. Task

The goals of this thesis are the development of a concept which allows programs to be executed securely, and a proof-of-concept implementation of the used approach. Secure execution means that neither the executed instructions nor the processed data are visible in memory at any time during execution, thus effectively protecting the program against cold boot and similar attacks on memory.

The chosen approach was inspired by Breuer and Bowen [14]. Here, the authors propose a “crypto-processor unit” (KPU), a CPU in which instructions and data enter and leave encrypted only. While the KPU concept does not seem to be practically feasible yet, the underlying idea is useful to us. If instructions and data appear decrypted only within CPU bounds, hence not in memory, the goal of secure execution would be met. The goal of this thesis can therefore be satisfied by creating a real world approximation of a KPU. Such an approximation consists of an interpreter which fetches encrypted instructions from memory into CPU registers, decrypts them solely using registers, and executes them. The data on which these

instructions work are fetched encrypted from memory, decrypted in similar fashion, altered by the instructions, encrypted again, and finally wrote back to memory. Thus, no plain text part of the interpreted program ever leaves the CPU bounds.

Because of the above mentioned constraints, the interpreter has to refrain from using main memory for managing the program execution. But, of course, somewhere memory space is necessary. The general purpose registers of the CPU are out of the question, as they are reserved for execution of the interpreter itself. Using the CPU cache would be a possibility, but it is very hard for the programmer to ensure that the cache's content is not somehow leaked to RAM. Fortunately, modern x86 CPU architectures provide quite a bit of additional memory, that is in form of the Advanced Vector Extensions (AVX) [15]. AVX adds 16 general purpose registers, each of 256 bit size. These registers can be used by the interpreter to hold the state of the executed program.

One difficulty stems from the fact that one has to perform an encryption algorithm using only CPU-registers. This especially means that the used encryption key has to stay secretly in registers all the time, otherwise the protection could be easily circumvented by reading out the used key from memory. Müller et al. [13] have shown that these requirements can be fulfilled: in their work, they demonstrate how the Advanced Encryption Standard (AES) [16] algorithm can be implemented without using memory. For that reason, their TRESOR Linux kernel patch is used for cipher key management and protection, as well as the technique they demonstrated for encrypting blocks of memory with AES. As it should be possible to use the interpreter in conjunction with the TRESOR hard disk encryption, running the interpreter should not weaken the overall security provided by TRESOR.

For a proof-of-concept it suffices if the interpreter supports only a simple bytecode language. The developed bytecode language uses only one data type, namely integers, and it provides only some simple arithmetic instructions to modify them. But for altering the control flow, the usual possibilities of an imperative programming language exist. There are unconditional and conditional branches, which allows the usage of conditional statements and loops. Additionally, there are call and return instructions, which allow the usage of subroutines, and therefore recursion.

To summarize, the goal of this thesis is to create an interpreter which executes programs securely outside RAM. For this, the memory attack resistant AES implementation of TRESOR is utilized, as well as TRESOR's key protection, and key management.

1.3. Related Work

Today, memory attacks on encryption keys, such as cold boot attacks or Firewire attacks, have been proven very viable [5, 6, 8, 9]. CPU bound encryption poses a successful software-based defense against memory attacks. The possibly most sophisticated work that implements CPU bound encryption on x86 is the afore mentioned TRESOR Linux kernel patch [13]. This thesis is using TRESOR as a foundation for implementing the interpreter. In particular, we will utilize TRESOR's key management, key protection features, and, last but not least, AES implementation. The preceding work to TRESOR was AESSE [11], which already implemented the AES implementation without using RAM, but had problems in regards to performance and compatibility with other programs. An alternative to TRESOR is "Loop-Amnesia" [12], which also provides a cold boot resistant AES implementation. Because Loop-Amnesia's implementation is limited to AES-128 and does not utilize AES hardware support, TRESOR was preferred as a foundation for this work. An early suggestion was "Frozen cache" [10], which reserves the CPU cache as key storage. As this makes the CPU cache unusable for every other application, the overall system performance would probably be slowed down considerably. On ARM based Android devices, a cold boot resistant AES implementation has also been developed [17]. This is only tangential to our work, as this thesis focuses on the x86 architecture, but might enable a port of our interpreter to the ARM platform in the future.

The above listed work is restricted in the sense that only the cipher key is stored securely. The main use case is to protect full disk encryption against memory attacks. In this work, we have another focus: to protect programs against memory attacks by executing them encrypted. Encrypted program execution has already been worked at, in different ways: Brenner et al. [18] presented how secure program execution could be possible by using homomorphic encryption [19]. 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 [20, 21], which would indeed protect programs against memory attacks, if encryption is implemented in a memory attack resistant way. However, [21] is restricted to ARM processors equipped with security hardware, while [20] relies on its own, special hardware architecture. A memory encryption solution explicitly designed to mitigate cold boot attacks against data is Cryptkeeper [22]. Unfortunately, on its own, Cryptkeeper poses no viable solution, because the cipher key is not stored in a cold boot safe way, e.g. within the CPU.

A possible future technology to solve all the issues surrounding secure program execution is Intel's Software Guard Extensions (SGX) [23]. 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 as well. 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 in 2013, it is still unknown when the first hardware devices supporting SGX will be released to the public.

1.4. Results

This thesis provides the concept and implementation of a bytecode interpreter executing programs without using RAM for code and data, thus effectively protecting them from memory attacks. This is achieved by encrypting all memory segments of the executed program, so that a memory attack can only find scrambled program content. Because the interpreter can not directly process encrypted instructions and data, the currently relevant parts of the memory segments are held decrypted withing the CPU's AVX registers. For encryption, the interpreter utilizes TRESOR, an AES implementation protected against memory attacks.

The working proof-of-concept implementation of the interpreter is targeting the x86 architecture, and is delivered in form of a kernel module compatible with all newer Linux kernels. The interpreter needs the CPU to support the instruction sets AVX and AES-NI, a requirement that most newer Intel and AMD CPUs fulfill. Alongside the interpreter, a bytecode language was developed. This language currently only supports some of the most essential programming concepts, namely integers, basic arithmetics, iteration and recursion. While the instruction set of the devised bytecode language is thus too small for real world applications, we show how the the language can be extended with more features in the future. Because programming in bytecode is uncomfortable, a complementary high level language, including the compiler which translates it to bytecode, is provided.

The most critical point of the concept is its security. The interpreter was evaluated in regards to several attack types: memory attacks, software attacks, and hardware attacks. Concerning memory attacks, we show that the interpreter can fulfill its goal and is indeed safe against those kind of attacks. Against attacks on the software level, the interpreter provides considerable security, protecting the confidentiality of executed programs even against attackers with root privileges. Restrictively, it has to be said that the interpreter only provides this security if the kernel's integrity is guaranteed. A flaw shows at the hardware level: a specially crafted DMA attack allows to circumvent the interpreter's protection. There are measures to mitigate that kind of attack though.

Besides security, a deciding factor regarding the applicability of the concept is performance. The interpreter was benchmarked 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 the interpreter, which is hardly surprising considering that these languages utilize native code execution. Between the interpreter and Python, the difference in performance is much smaller, with Python being faster than the interpreter by an average factor of 4. This is the price paid for security, as the interpreter spends a substantial amount of time on encryption – in fact, up to 83 % of the overall execution time. Thus, a topic for future work is how the interpreter's performance can be further increased to extend the applicability in practice.

1.5. Outline

Chapter 2 gives some helpful background information in order to understand the topics treated in this thesis. In particular, section 2.1 introduces what an "interpreter" is. In section 2.2, the encryption algorithm used in this work, AES, is presented. Section 2.3 gives a detailed overview of the above mentioned TRESOR encryption system, which will occur frequently throughout this thesis.

In chapter 3, the interpreter's design and implementation are described. Section 3.1 introduces the different parts the interpreter consists of, and how they interact with each other. Section 3.2 depicts where and how the interpreter manages the state of an executed program. In section 3.3, we discuss how TRESOR's encryption algorithm was adapted to fit the needs of the interpreter, and how encryption is applied to interpreter data. With the previous information, section 3.4 shows the steps the interpreter goes through while executing a program. The last section 3.5 is about the developed bytecode language. The different bytecode instructions are covered, a short example program is described, and limitations of the language are discussed.

The interpreter implementation is evaluated in regards to several aspects in chapter 4. We discuss compatibility in respect to software, hardware and other programming languages is in section 4.1, performance in section 4.2, the correctness of the AES implementation the section 4.3, and finally security against several kinds of attack in section 4.4.

The last chapter 5 contains a discussion of limitations of the devised concept in section 5.1, as well as ideas for further developments of the interpreter in section 5.2. And finally, some concluding considerations about the applicability of the interpreter concept are made in section 5.3.

1.6. Acknowledgments

First, I would like to thank my advisors, Tilo Müller and Michael Gruhn, for letting me work on this challenging topic. I really learned a lot from it.

Then, I would like to thank my parents for the continued support they gave me throughout my studies, and my grandfather for being an inspiring example. Finally, a big thank you to my friends Leonie and Pablo, who gave me quite a few invaluable language corrections.

2 BACKGROUND

This chapter gives some introductory knowledge about some of the topics which appear throughout this thesis. In section 2.1, we explain the concept behind an interpreter, and discuss different approaches to implement them. In section 2.2, we introduce the used encryption algorithm AES. As this work does not need to implemented AES itself, but rather uses an existing implementation, the mathematical details will be spared. Section 2.3 details the TRESOR encryption system, which, among other things, provides the needed AES implementation.

2.1. Bytecode Interpreters

To start, a short classification of what a "bytecode interpreter" actually is: Essentially, an *interpreter* is a program which executes *other* programs. A real CPU can only execute programs that are given as machine code. A compiler generates an executable in machine code from source code written in a programming language. Such an executable is bound to a specific hardware architecture; it can only be executed on the target platform it was compiled for. An interpreter abstracts from the hardware platform by providing an execution environment where programs can be run in, without the need of compiling to machine code. This has the advantage that programs written in an interpreted programming language can be run on any hardware platform, provided there is an interpreter implementation for this platform. Usually, programs to be interpreted are given either directly in source code, like in a scripting language like Perl, or in an intermediary *bytecode*, for example in the languages Java, or C#. In the same way that programs are compiled to machine code to run on a specific architecture, they can also be compiled to bytecode to run with a specific interpreter. Bytecode can be executed directly by the interpreter without further transformation, and thus, bytecode can be regarded as the "machine code" of an interpreter. This is the simpler version; frequently, interpreters compile bytecode at runtime further to native machine code which can then be executed efficiently on the hardware again. This is known as "*just-in-time-compilation*" [24]. The term "bytecode" stems from the fact that interpreter instructions are often encoded in numerical opcodes with a size of one byte.

```
char bytecode[] = {0x1, 0x2, /*...*/};
int pc = 0;

while(1) {
    switch(bytecode[pc++]) {
    case 0x1:
        /* execute instruction A */
        break;
    case 0x2:
        /* execute instruction B */
        break;
    }
    /*...*/
}
```

Figure 2.1.: switch dispatch

```
void *jump_table[] = {
        &&instr_A, &&instr_B, /*...*/
};

int bytecode[] = {0x1, 0x2, /*...*/};
int pc = 0;

goto *jump_table[pc];

instr_A:
     /* execute instruction A */
     goto *jump_table[bytecode[pc++]];
instr_B:
     /* execute instruction B */
     goto *jump_table[bytecode[pc++]];
/*...*/
```

Figure 2.2.: indirect threading

The basic principle of how an interpreter executes a bytecode program is quite simple: The current position in the executed program is marked by the program counter (in section 3, the program counter is referred to as the *instruction pointer*). The interpreter extracts the opcode at the current position, decodes the instruction which belongs to the opcode, executes said instruction, and advances the program counter to the next instruction. Figure 2.1 shows an example of how this algorithm could be approached in the language C. Note that, for brevity, there are only two instructions shown, and the instruction implementations are also omitted. The bytecode to be executed is given as an array of opcodes, the program counter is an index into this array. Decoding the opcode into an instruction is done with a `switch` statement, in which each individual `case` block represents an instruction. Instruction decoding and execution is repeated endlessly by a loop, until an instruction breaks the loop and terminates execution. This example shows only linear control flow, in each iteration, the program counter is increased by one, to point to the next bytecode instruction. Of course in practice, there are also *jump instructions* which can change the program counter, and thus the control flow, arbitrarily. Because a switch statement is controlling the threading of instructions, this technique is known as *switch dispatch* [25].

There are many different techniques for implementing an interpreter. The difference mostly lies in the way the instructions of the interpreted program are "threaded" one after another, which has an impact on performance. Another approach to interpreting is *indirect threading*, which is also the approach our bytecode interpreter uses. An example for how a C implementation to this approach could look like, is presented in figure 2.2. You can see that the `while` loop and the `switch` statement have disappeared. Instead, each instruction now directly jumps to the following instruction. This is done with a *computed goto* statement [26]. After a computed goto, the control flow continues at the label specified after the goto. Each interpreter instruction begins with a label, and all instruction labels are listed in the jump table. The jump table is an array of pointers to instruction labels. Each opcode is now taken as an index into the jump table, indexing the label of the instruction the opcode is corresponding with. After an instruction has finished executing, a lookup in the jump table is performed with the next opcode, which returns the label of the next instruction. `goto` then jumps to this label, and the next instruction is executed. It should be noted that the computed goto statement does not belong to the ANSI C standard, but is "only" a GNU C extension. However, this does not have to bother us, because our interpreter will be implemented in assembly language anyway, where the "computed goto" can be easily implemented by jumping to an address contained in a register, which is called an *indirect jump*.

There is also *direct threaded* interpretation. This technique is similar to indirect threading, but instead of opcodes, the program consists of the addresses to the instructions to be executed. Therefore, an instruction can directly jump to the next instruction, and a jump table lookup for the address is no longer required, which is also why direct threading is a bit faster than indirect threading.

The remaining question is which interpreter technique is preferable over the other. The "switch dispatch" approach is executed quite a bit slower than direct threading (in the following, no difference will be made between indirect and direct threading, because the analysis applies to both of them). Ertl and Gregg [25] found that interpreters with direct threading ran up to twice as fast as those interpreters who used switch dispatching, at least in some cases. Now, to find out the reason to this performance difference, we have to take a small excursus into how CPUs execute instructions internally.
Modern CPUs do instruction pipelining, that is, they already execute the next few following instructions even before the current instruction is finished. This of course brings a performance boost, because the CPU processes multiple instructions in parallel. The problem to instruction pipelining are conditional branches in the code. The CPU can only pipeline one of the two possible program branches, but it does not know which of the branches will be chosen at the time the branch is really executed. The CPU speculatively pipelines one of the branches; if it turns out that this branch was not the actually chosen branch, the whole pipeline has to be discarded, which has a negative impact on performance. To reduce the occurrence of such pipeline flushes, the CPU tries to predict if a branch will be taken or not, and if it is an indirect jump, to which address the jump will go to.
Now we return to the two interpreting techniques. In switch threading, the next instruction to jump to is chosen at a single location: the switch statement. Because this jump can jump to all the possible interpreter instructions, it is very hard to predict for the CPU at which instruction this jump will really end up. In fact, nearly all instructions will be predicted wrong [25], causing a pipeline flush every time a new interpreter instruction is executed. Direct threading performs quite a bit better in this respect: because every interpreter instruction has its own goto at the end, the CPU can, over time, analyze which interpreter instructions are more likely to be executed after particular interpreter instructions. This allows the CPU to predict most jumps correctly. Ertl and Gregg [25] talk about over 90 percent accuracy for CPUs which have "two-level indirect branch predictors", which are, by this time, most of today's CPUs [27].

In this section, we classified the term "bytecode interpreter" and showed how different interpreting techniques can be implemented. This section also provided the rationale for choosing the indirect threading approach over switch threading for our interpreter.

2.2. The Advanced Encryption Standard

A recurring topic throughout this thesis is encryption. The encryption algorithm we apply is the well known Advanced Encryption Standard (AES) [16]. In this thesis, we use encryption more or less as an opaque process, that is, we have an existing implementation at hand that will be used by us. As we do not need to implement AES ourselves, the reader does not need to be familiar with the complex mathematical details behind AES, so we will spare them in this section. Instead, we are going to focus on the general properties of AES that are important to know while reading this thesis.

Historically, the Advanced Encryption Standard was standardized by the National Institute of Standards and Technology (NIST) in 2001. It originated from the Rijndael cipher, devised by the Belgian cryptographers Joan Daemen and Vincent Rijmen, which was chosen among fifteen competing ciphers in the standardization process. The AES cipher is deemed very secure, and, up to today, is considered unbroken. Even the best known attack [28] is of no practical relevance. Besides being secure, AES has the advantage that it can be run pretty performant in hard- and software. With the AES instruction set (AES-NI) [29], there exists a widely available hardware implementation of AES on Intel and AMD CPUs which allows speeding up the encryption process significantly. The AES implementation used by our interpreter utilizes AES-NI, and thus its availability is a prerequisite to running the interpreter.

To categorize, AES is a symmetric block cipher. Characteristically, a symmetric cipher shares the same cipher key for encrypting a plain text and decrypting the cipher text. In contrast, asymmetric ciphers use two different cipher keys: a private and a public key, where the private key is used for decryption, and the public key for encryption. Symmetric ciphers can be further divided into stream ciphers and block ciphers. Stream ciphers can encrypt a data stream of arbitrary length, whereas block ciphers encrypt exactly one block of a specific size at a time. AES has a block size of 128 bits.

A further property of ciphers is the length of the cipher key. In general, the longer the used cipher key, the more complex it is to break the cipher. AES specifies three variants, AES-128, AES-192 and AES-256, with different key sizes of 128, 192, and 256 bits.

The AES encryption algorithm is divided into several processing rounds, namely ten rounds for AES-128, twelve for AES-192, and fourteen for AES-256. In each round, multiple transformations are applied to the output of the previous round, which is called the encryption state. The input of the first round is the plain text and the output of the final round is the encrypted cipher text. From a cipher text, the original plain text can be recovered by reapplying the encryption rounds, just in reverse order. Here, we will omit what the actual mathematical transformations are. Important is just that at one point in each round, the current encryption state is chained to a 128 bit round key by xor'ing. Because the round keys are generated from the cipher key, this operation is where the encryption state is made dependent on the cipher key. The round keys are generated before the actual encryption rounds take place, through a process called the "Rijndael key schedule". There is one round key for each encryption round, plus one additional round key that is initially applied to the plain text. Because the key schedule is invertible [30], the cipher key can be also be generated backwards from round keys. This has the consequence that not only the cipher key has to be kept secret, but also the round keys, as we will see later.

Although this being only a short overview, it should give you enough insight to understand how AES is used throughout the thesis. In the next section, the TRESOR Linux kernel patch will be introduced, that, among other things, provides the mentioned AES implementation of the interpreter.

2.3. TRESOR

This work is based on the TRESOR Linux kernel patch [13]. Because the bytecode interpreter makes strong use of several features that TRESOR provides, it is worthwhile to introduce it separately, and in depth. TRESOR is a work by Müller, Freiling, and Dewald, published in 2011. The contribution of TRESOR is the efficient implementation of AES in a way that is resistant to memory attacks, while still running efficient. TRESOR is currently distributed for Linux kernel versions 3.6.2 and 3.8.2. Because TRESOR integrates itself into the kernel's cryptographic API, the AES implementation is in principle compatible with every application supporting the crypto API, such as the hard disk encryption software dm-crypt [31].

The predecessor of TRESOR was AESSE [11] in 2010. AESSE, like TRESOR, provides an AES implementation that is protected against attacks on the cipher key in RAM. In conventional disk encryption solutions, such as dm-crypt or TrueCrypt [32], the cipher key is continuously stored in RAM, over the whole life time of a system. This makes common encryption programs like the above mentioned ones susceptible e.g. to cold boot attacks [5]. Now, the basic idea to protect encryption against memory attacks is that if RAM is too insecure to store the cipher key, the cipher key must be stored in another place that can not be read out by a cold boot attack. For AESSE, this other place are the CPU's SSE registers, because CPU registers do instantly lose their values if power is switched off, thus they can not be read out after a reboot of the system. The drawback here is that any CPU registers that contain the cipher key must be reserved for exactly that purpose. Any other application that uses the SSE registers may, accidentally or not, overwrite some of the key holding registers, which in turn leads to corruption of the encrypted data. This is also the main disadvantage of AESSE: special measures must be taken to ensure that the SSE registers are exclusively used by AESSE, which breaks compatibility with any program that uses the SSE registers as well. Because it is relatively common that applications utilize the SSE registers, AESSE's usage in practice is limited.

With TRESOR, the applicability increases significantly. There are two major improvements over AESSE: First, instead of the SSE registers, the CPU debug registers hold the cipher key. This has several advantages. The frequently used SSE registers are left free to use for other applications, which results in TRESOR being compatible with those applications. In contrast, the debug registers are rarely used by most applications, only by specialized software, first and foremost debuggers. But even debuggers are not dependent on the debug registers, as their main purpose, namely break points, can be compensated in software well. Whereas the SSE registers are free to access by every user space application, debug registers are a privileged resource,

which means that innately, only ring 0, the kernel, can access them directly. This makes them much easier to protect from external access than the SSE registers. The second improvement over AESSE concerns performance. The AES implementation of AESSE is comparably slow, because the transformations required by AES can not be performed as efficient in SSE registers as in memory. In contrast, TRESOR is utilizing the afore mentioned AES instruction set [29]. These instructions are implementing the AES transformations in hardware, which altogether yields performance on par with conventional AES implementations.

Although only the kernel may *directly* access the debug registers, some further measures are required to completely isolate them from user space. Again, this is necessary because it must be ensured that the cipher key stays both secret and unchanged. Linux lets user space applications access the debug registers through the `ptrace` system call. As it is distributed as a kernel patch, TRESOR is able to change the behavior of system calls. Thus, if a debug register is requested by an user space application, `ptrace` has been changed to return `0` instead of the real register value. An attempt to write to a debug register with `ptrace` now results in the `-EBUSY` error code. Another issue with the debug registers arises with the operating system performing *context switching*. When the operating system switches to another process, the context of the current process is saved to memory, so that it can be properly restored when this process runs the next time. The debug registers are part of the process context, so after a context switch, the operating system would write the cipher key to memory. To prevent that, TRESOR patches the standard routines to access the debug registers from within the kernel (`native_get_debugreg` and `native_set_debugreg`).

In the previous section we saw that, besides the cipher key, AES uses several round keys that have to be generated before the encryption can take place. Because the round keys are derivations of the cipher key, it is required that the round keys are kept out of memory as well. TRESOR holds the round keys in the SSE registers. Instead of saving them permanently there, they are generated on-the-fly for each encryption operation, and deleted again afterwards. But this alone does not guarantee that the round keys do not leak to memory, because, again, there is a problem with context switching. The SSE registers containing the round keys are also part of the process context; if the round keys should not be saved to memory by a context switch, it must be ensured that no context switch occurs during the period of time the SSE registers contain the round keys. TRESOR solves this by running each encryption operation *atomically*. When the CPU encrypts a block of data, no other process is allowed to interrupt until encryption is finished, not even the operating system. Before the atomic section is ended again, the SSE registers are cleared, so that a following context switch does not leak any round key to memory. In section 3.4 it is described how exactly such an atomic section can be implemented within the Linux kernel.

To provide a rough understanding how TRESOR implements the AES encryption process, a short sketch is provided here. The description is composed in a textual manner, to spare you from the actual Assembler implementation. We are only regarding AES-256, the other variants work similarly. Figure 2.3 shows you TRESOR's register usage in the AES-256 case. Here, `rstate` is the register that holds the current encryption state, and `rhelp` is a helper register holding intermediary values. First, 16 bytes of plain text to encrypt are copied to the `rstate` register. Then, the cipher key is loaded from the debug registers (`db`) to round key registers 0 (`rk0`) and 1 (`rk1`). The lower 8 bytes of `rk0` are filled from `db0`, the upper 8 bytes from `db1`. The same happens for `rk1`, but with `db2` and `db3` instead. Round key 0 is only needed initially, thus it is applied to `rstate` by xor'ing. The next step is to sequentially generate round keys 2 to 14 with the AES key schedule. Each round key is generated from the previous two round keys, starting with round keys 0 and 1 in `rk0` and `rk1`, thus the cipher key. In the last key schedule round, register `XMM2` with round key 0 is overwritten with round key 14. This can be done because round key 0 was already previously applied to the plain text, and is no longer needed at this point. After round key generation, the 14 AES encryption rounds can be performed. In each round, the round key register is applied to the current encryption state, writing the resulting encryption state to `rstate`. Finally, after the fourteenth encryption round, `rstate` contains the cipher text, which is then copied back to memory.

Another yet undiscussed question is how TRESOR loads a cipher key to the debug registers in the first place. As our bytecode interpreter relies on TRESOR for the key management, this problem is also of further interest to us. The challenge here is to ask the user for a password without this password touching memory. It turns out that this is ultimately not possible, and the most secure way to fill the debug registers with the cipher key is during boot time. Immediately after the system is turned on, TRESOR displays a prompt in which the user has to enter his encryption password. TRESOR then derives the cipher key by

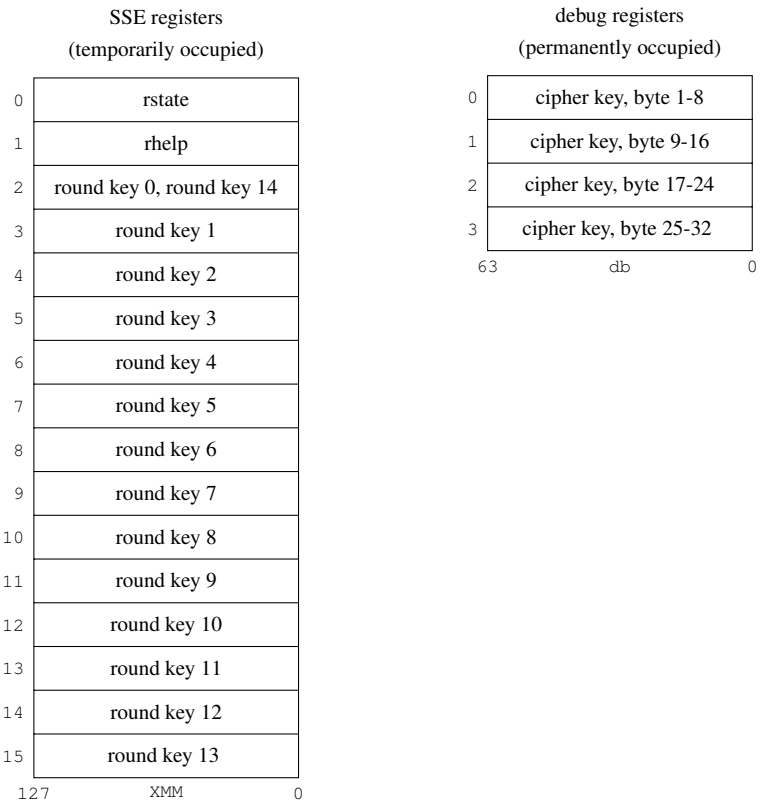


Figure 2.3.: TRESOR's register usage for AES-256.

hashing the password 2000 times, with the SHA-256 algorithm. Finally, the cipher key is copied to the debug registers, and the boot process continues. For reading in the password and computing the cipher key, memory has to be used. When this is done early during booting, this is ultimately of no issue, because at this point, only kernel space is existing, and the critical memory locations are reset before user space comes alive. It is also highly unlikely that someone performs a cold boot attack *while* the user is starting up his system.

A disadvantage of TRESOR's key management is that it supports only one active cipher key at a time. This is of course because the cipher key is stored in the debug registers, and those provide only 256 bits space, which is just enough for the cipher key of AES-256. But that forbids for example to simultaneously use two disk partitions encrypted with different keys. Another inconvenience for the user is that to change the cipher key, the whole system has to be rebooted. In the two disk partitions example, this becomes annoying if the two partitions are to be used one after another: to mount the other partition, a reboot has to happen. However, this is a non-issue in the common case of full disk encryption, where only one cipher key is in use. The password must be entered one time at boot, and the user is good to go.

To conclude, TRESOR is a Linux kernel patch providing an AES implementation that is resistant to cold boot attacks. The cipher key is stored outside RAM, in the debug registers. TRESOR patches the kernel to deny access to them. The AES implementation itself utilizes hardware acceleration through AES-NI, resulting in good performance. Encryption state and round keys are stored in the SSE registers. To prevent leaking round keys to memory through context switching, encryption of blocks is run atomically. TRESOR generates the cipher key from the user's password at boot time. Finally, as TRESOR poses the basis of our bytecode interpreter implementation, the properties described in this section will reoccur further throughout this thesis.

3 IMPLEMENTATION

In the following chapter we describe design and implementation of the interpreter. While implementing our interpreter, we have to keep our two security policies in mind:

- do not use main memory for any sensitive data, as memory is considered untrusted
- do not weaken the given security of a system provided by TRESOR

We will solve the first challenge by enforcing that any data is encrypted before it hits memory. The second task can be fulfilled by ensuring the confidentiality of the TRESOR cipher key during interpreter runtime.

The first section, 3.1, introduces the different parts the interpreter consists of, and how they interact with each other. Section 3.2 depicts where and how the interpreter manages the state of an executed program. In section 3.3, we discuss what changes were made to the encryption algorithm in respect to TRESOR, and how encryption is applied to interpreter data. Section 3.4 characterizes the steps the interpreter has to go through to execute a bytecode program. The last section, 3.5, is about the developed bytecode language. We cover the different bytecode instructions, describe a short example program, and discuss the limitations of the language.

3.1. General Interpreter Composition

In this section, we showcase what the different parts the interpreter consists of are, what they do, and how they work together. We do this by walking-through a programs 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-like 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 figure 3.1.

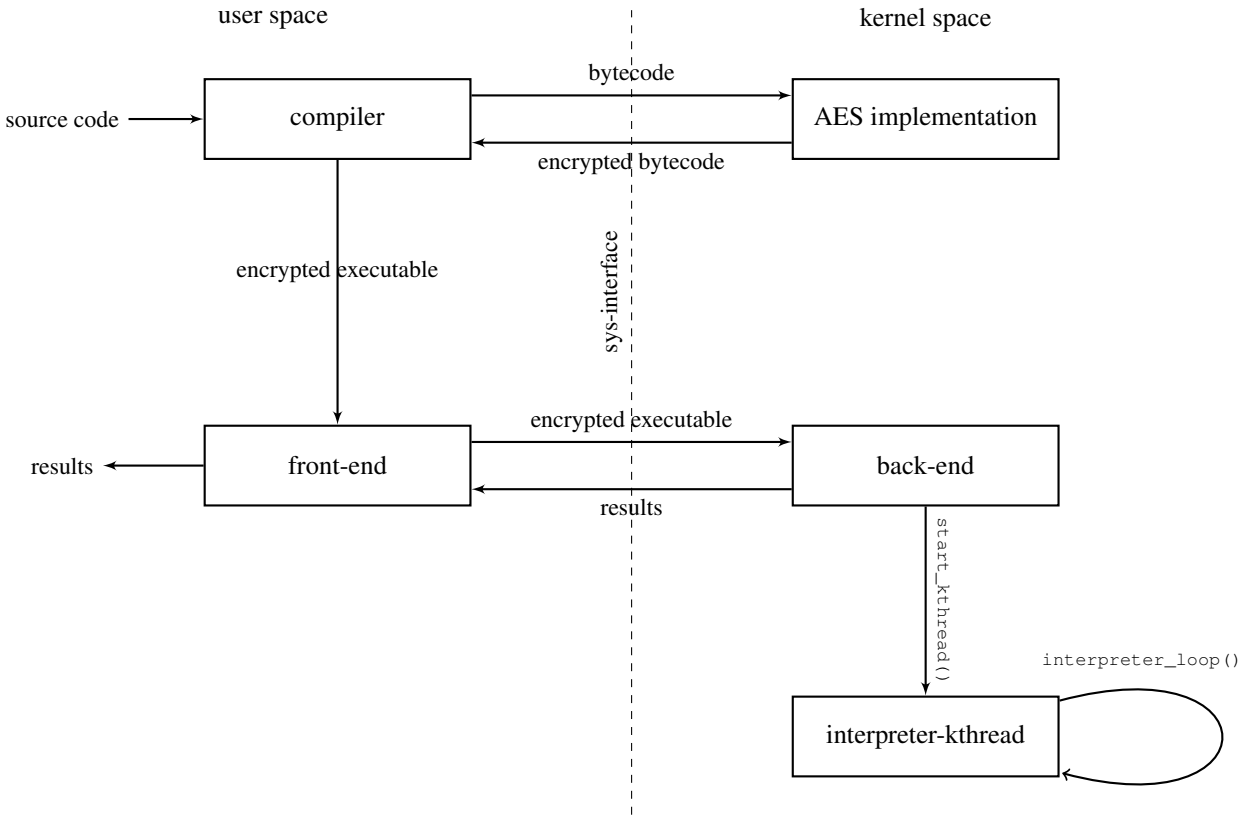


Figure 3.1.: The interpreter is separated into compiler and front-end in user space, and back-end with the AES implementation in kernel space. The different parts communicate over the kernel's sys-interface. On program execution, the back-end starts a kernel thread running the interpreter loop.

At first, the Linux kernel has to be booted up. This is actually an important step, because at this point, TRESOR asks for the user password. During the system's life time, every program created will be encrypted with this password, and every program the interpreter executes will be decrypted with this password. After a password was chosen, 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*", and fittingly, the corresponding file extension is `.scll`. As one can guess by the name, this language is strongly based on the language C. However, it is stripped off many of C's features because it is limited by the capabilities of the underlying bytecode language. This means that the language lacks many usual features such as arrays or global variables. Nevertheless, it is much easier to work with than programming directly in bytecode. For reference, the grammar of SCLL is given in section A.1.

A finished program is passed to the compiler to translate it to bytecode. The compilation procedure itself is not covered in this thesis, as it is not of great importance to the overall process. After compilation is complete, the bytecode is not yet ready to be executed; to meet our goal of secure execution, it has to be encrypted first. Encryption needs the cipher key, and the cipher key is stored in the debug registers, which in turn can only be accessed from kernel space. Thus the encryption process needs to happen in kernel space. 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, at `/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 cipher key. After getting back the encrypted bytecode, the program is then output 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 named `/sys/kernel/bispe/invoke`. Alongside the program, additional information is passed to the back-end, e.g. execution settings and buffers providing space for execution results. The front-end then blocks until the back-end has finished execution of the program.
Before execution can begin, the back-end first has to initialize the execution environment; one can compare this to the process an operating systems loader has to go through whenever a new process is started. Most notably, this means allocating the different memory segments the interpreter uses, which 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 section 3.2. After initialization, it is finally time to start the execution: 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, this makes clean signal handling easier. 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 is extensibility: Currently, only one interpretation can be done at once. If interpretation is already done in a separate thread, it will be 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 section 3.4, and for now, it suffices to know that this is the point where our program gets actually executed. 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.

We had a look at the different stages a program has to go through on its way to execution. In the next section, we describe how the interpreter memory layout is structured.

3.2. Interpreter Memory Layout

In the last section, the different memory segments the interpreter uses were already briefly touched on. This section details how the interpreter organizes the executed program's memory, and in which way the encryption interacts with the data.

Before the memory segments are described, a few preliminaries have to be clarified first. 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. This is in contrast to a register based interpreter, which simulates multiple registers which instructions use for operands and results. The reason, why a stack based interpreter was chosen over a register based one is that the former is easier to implement and the bytecode instruction set is simplified. On the other hand, register based interpreters have been found to offer better performance [33]. The here presented concept is expected to be transferable to a register based interpreter as well.

The interpreter uses a word size of four byte for every instruction and every data element. An instructions consists of a four byte opcode and a four byte argument, if the instruction specifies one. Obviously, one could fit the instruction set of the interpreter in less than four byte. Even one byte would suffice for the 22 instructions the interpreter currently has. But the usage of the same word size for instructions and data elements allows a few important simplifications in the way data is accessed. The unit by which the interpreter accesses memory is a row. A row consists of 16 byte. Every time the interpreter reads from memory, he has to read in a full row, even though the requested data is only of word size. This is 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. Thus, it is impossible to only read in a single word, because for this word to be decrypted, the full row, in which the word is contained, is needed.

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. We will see why this is useful later on. We now describe for what each segment is used.
As the name tells you, the *code segment* stores the code of the program. Before the execution starts, the encrypted bytecode is moved to this segment. Additionally, there is a pointer pointing to the address of the instruction which is executed next. Currently, this pointer resides unencrypted in memory. The reason for this is that the instruction pointer changes in a very foreseeable manner, normally 4 bytes at a time, namely when instructions get executed sequentially without any jumps. If the pointer would be encrypted, it might be possible to extract information about the cipher key from that. Unfortunately, an adversary now has the possibility to get meta information about the executed program. He can not monitor exactly which instructions are processed, but how the control flow moves throughout execution. A possible fix to this problem is to encrypt the pointer, but to pad it with random data before. Thus, the encrypted pointer data has a different, unforeseeable structure independent of the control flow of the program. This solution was currently not implemented, and is an improvement for the future.

Intermediate data is stored in the *operand stack segment*, with a stack pointer always pointing to the top element of the stack. The operand stack behaves just like a conventional stack, i.e. it grows by inserting elements on top of the stack and shrinks by removing elements from 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 the load and store instructions, which can transfer data between the operand stack and variables. For example the instruction `add`, which adds two integers together, takes the top two elements from the operand stack, adds them, and pushes the result back on the stack, where other instructions can process it further. Furthermore, 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. An element in the current subroutine frame should be accessible without having to remove every element up to the element from the stack first. That is why, in contrast to the operand stack, every element on the call stack can be freely accessed in respect to the call stack pointer.

We now take a look at how data from the segments can be used by the interpreter, although it rests encrypted in memory. Encrypted data can not be processed by instructions directly, without intermediate decryption. For this to work, a fully homomorphic encryption would be necessary. There is indeed research in this direction [19], but these efforts are not practically feasible yet. So the data is somewhere needed in an unencrypted form first, before it can be processed. And this place may not be main memory itself, as forbidden by our security policies. That is why 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 up to get processed. Instructions can now always 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 first. This can simply be done by setting the lowest four bits of the pointer to the element to zero. This is possible because, as previously mentioned, all segments are aligned to 16 bytes. Then, 16 byte from the pointer calculated this way are copied from memory to the corresponding row register. The register gets decrypted and the data is now, in principle, ready to access (before a bytecode instruction can actually use it, the element needs to be extracted from the register row first to a general purpose register first, but this is easily done with a few Assembler instructions).

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 now 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.

The depicted procedure is largely the same for the code and the call stack segment. Note that the row register of the code segment does not need to be re-encrypted back to memory, as the content of the code row register is never modified during runtime. That is because the code segment is read-only, and there exist no bytecode instructions which could alter it. If the instruction pointer exceeds the current row, it suffices to just overwrite the old data in the row register with the new data. For the call stack segment, a simple caching mechanism is also in place. Normally, if an element on the call stack needs to be accessed, the row, the element is contained in, gets loaded from memory to the call row register. However, if the row register already contains this exact row, because it was loaded previously, this row should not have to be loaded again. This is implemented utilizing an additional pointer which points to the memory address of the row the call row register currently contains. When an element is to be loaded, it is first checked if the row of this element is already loaded to the call row. If so, the element can just be extracted, saving the overhead of decrypting the full row again.

In this section we saw what the different memory segments of an interpreted program are used for. We described how excerpts of the memory are stored in row registers within the CPU, and how data is moved decrypted to and encrypted from these registers. The next section gives more information on how the encryption process takes place.

3.3. Encryption Scheme

Previously, the topic of encryption was mentioned, but an explanation, how encryption is incorporated in the overall system and in which way it is actually applied, was yet omitted. We will go into more detail about that in this section.

As mentioned before, the interpreter utilizes the AES encryption and decryption routines provided by TRESOR. In particular, the interpreter uses only AES-256 (whereas TRESOR also offers the other AES versions AES-128 and AES-192). That means that the used cipher key has a length of 256 bits. As already introduced in section 2.2, AES-256 uses 14 round keys, one for each encryption round, plus an additional round key that is initially applied. Before the actual encryption can take place, the different round keys have to be generated from the cipher key. In a normal AES implementation, the round keys are simply stored in RAM, but because the round keys are derived from the cipher key, they, like the cipher key, have to be kept from leaking to memory, to concur with the security goals. TRESOR originally solved this by saving the round keys in the SSE registers `XMM2` to `XMM15` instead of memory. Because the SSE registers can not be permanently allocated for the round keys, round key generation has to be done on-the-fly for each individual block. During the encryption process, TRESOR saves the current AES state in `XMM0`, and `XMM1` is reserved as a helper register, holding intermediate values. This means that TRESOR allocates every single one of the 16 SSE registers for encryption, leaving no spare registers for the interpreter to use. But as the interpreter needs some SSE registers for itself, namely the row registers containing parts of the interpreted programs state, the register distribution has to be changed compared to TRESOR. Fortunately, with the introduction of the Advanced Vector Expansions, the size of the SSE registers was increased from 128 bits to 256 bits [15]. 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 3.2 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 crypto-routines of the interpreter are implemented in `km/bispe_crypto_asm.S`. Programmatically, most of TRESOR's AES code could be carried over, but a few changes were made. First of all, the AES-128 and AES-192 encryption functionality was removed, because the interpreter simply has no need for it. Some modifications that had to be made reflect the changed way round keys are stored: it is now necessary to extract the round key to a helper register before an encryption round can take place. This is due to the fact that AVX instructions can not address the upper half of an AVX register with a register prefix. One can only address a whole register with the `YMM` prefix, or the lower 128 bits with the `XMM` prefix. And because half of the round keys are indeed stored in the upper half of the registers, they have to be extracted to an accessible (XMM) register first. Another change is that the interface was extended. TRESOR just has routines for encrypting and decrypting blocks of memory (`tresor_encblk_256` and `tresor_decblk_256`). The crypto module of the interpreter now offers routines to encrypt memory in cipher block chaining mode (`bispe_encblk_mem_cbc`), as well as encrypting a register in place (`bispe_encblk`).

Furthermore, the code generating the AES round keys was separated from the encryption routines and made accessible from the outside (`bispe_gen_rkeys`). This allows us to generate the round keys only once for successive encryption passes, saving work. Because the round keys are now stored in different registers then before, the round key generation routine was also modified to reflect those changes. This involves that the algorithm now only works on the three helper registers, plus the state register, instead of using all round key registers, like in TRESOR. After generation, the round key is inserted into the destination YMM register. Again, this has to be done because the upper half of an YMM register can not be individually accessed by instructions. The changes to the algorithm only involved modifying the registers on which the instructions work, so that the algorithm should still work correctly. This claim is verified in section 4.3.

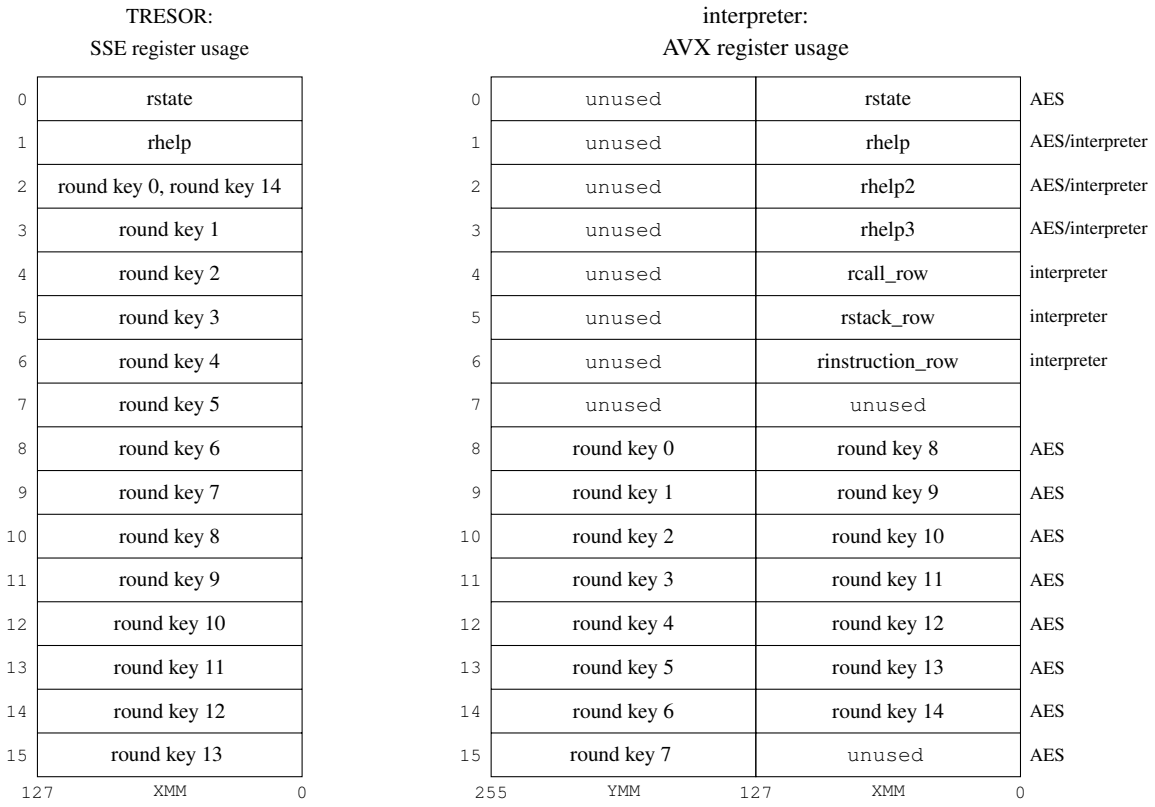


Figure 3.2.: The interpreter modifies TRESOR's register distribution to make room for the program state.

You might notice from figure 3.2 that the new register distribution leaves the upper half of the first seven AVX registers unused. One could ask why this space is not used for round keys too, so that every round key resides in the upper half of a register, yielding an overall cleaner register layout. The reasoning behind that is rooted in the subtleties of the AVX instruction set. Whenever an AVX instruction writes only to a `XMM` register, not to the whole `YMM` register, the upper 128 bits of this register are zeroed out [34]. There are still legacy supported SSE instructions which do not show this behavior, that is, they leave the upper half intact when using an `XMM` register. But mixing new AVX instructions with old SSE instructions induces a heavy performance penalty, according to Fog [35]. We can easily avoid this penalty by using only AVX instructions, but this forces us to leave the upper halves of those `YMM` registers unused whose `XMM` part is to be used separately. This affects the state register, all of the temporary helper registers, and the three row registers.

The next topic about encryption we discuss is how the cipher mode is applied. On their own, block ciphers encrypt only one block of data at a time. The used cipher mode determines over how consecutive blocks of data are processed, deciding over confidentiality and authenticity of the encrypted data. The simplest cipher mode, electronic codebook (ECB), just encrypts every block of data independent from each other. As a consequence, patterns appearing in the plain text are preserved in the cipher text. In our use case, (bytecode) programs consist of well-structured data, which means that this structure carries over to the ECB encrypted program. Thus, applying ECB to encrypt the interpreter's programs would not provide optimal confidentiality of the encrypted data. Instead, the interpreter uses *cipher block chaining* (CBC) [36] as its cipher mode. Cipher block chaining makes each block dependent on all previous blocks by chaining them together. Additionally, a random *initialization vector* (IV) is used to make the encrypted data unique, even if encrypted with the same cipher key. If a good source of randomness is used for the IV, CBC is widely considered secure, and thus the interpreter uses it for each of the three memory segments. There are further cipher modes that also guarantee authenticity of the encrypted data, but they can be much more complicated to implement, so the interpreter limits itself to CBC. If it turns out other cipher modes are needed, they can still be integrated in the future.

CBC works by xor-ing the plaintext with the ciphertext of the previous block before encryption. That means for the encryption of the first block, the initialization vector is needed, because there is no previous block to xor' with. The IV has the same size as one block, so, in this case, 128 bits. The interpreter generates these vectors at different times depending on the memory segment. For both stack segments, the IV is created at runtime, directly after the segments were allocated, that is, before program execution is started. 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 and it stays the same for the executable until a recompilation occurs. Before sending the bytecode to the kernel module for encryption, the compiler writes 128 random bits in front of the bytecode, this time read from `/dev/urandom`, which ultimately also uses the Linux kernel's PRNG. The kernel module then encrypts the code in CBC mode, using the routine `bispe_encblk_mem_cbc`, and the encrypted code, including the IV in front, is written to the executable file.

The security of the CBC mode stands and falls with the randomness of the initialization vector, that is, how predictable it is. There seem to be differing results on the cryptographic quality of the random numbers output by the Linux kernel. For example, Dodis et al. [37] have proven that Linux PRNG is not "robust" (in respect to their self-defined robustness criterion), whereas Lacharme et al. [38] found out that "the design of the current version of the PRNG allows it to reach a good level of security". More positive results were found in a study issued by the Bundesamt für Sicherheit in der Informationstechnik [39]. All in all, it can be argued that the Linux PRNG is suited well enough for generating random data necessary for initialization vectors.

The implementation of CBC mode for the memory segments during runtime is, in all but one case, trivial. To load an encrypted row from any segment into a row register, move 16 byte from the row pointer to the appropriate row register, decrypt it in place using the `bispe_decblk` routine, and xor' it with the memory block 16 bytes before the row pointer. To save a decrypted stack segment row to memory, xor' it with the previous row, encrypt it in place, and move it to memory. Notice that this is only possible because the stack row register contains always the *last* (the topmost) row of the stack segment, so that no rows come after it.

In CBC mode, a block depends on any block coming before it in the block chain. If any but the last block of the chain gets re-encrypted with different data, every block coming after this block has to be re-encrypted too, up to the end of the block chain. This is what creates the non-trivial case, too: 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. This sounds like it would create a lot of overhead, but in practice, when a row gets saved to memory, most of the time it already lies close to the end of the segment, so that only few blocks actually have to be re-encrypted. This stems from the fact that call stack write operations always target the stack frame of the currently running function, which is always the topmost stack frame. For the exact implementation of the re-encryption operation, we refer to the Assembler macro `encrypt_reg_cbc_chain` in `km/bispe_state_macros.S`.

In this section we have seen how the encryption routines from TRESOR can be used by the interpreter and what had to be adapted to fit the needs of the interpreter. We discussed the generation of initialization vectors for cipher block chaining mode, and how this mode is applied to the different memory segments. Now that we have knowledge of the memory layout and how encryption works, we can take a look into the individual steps the interpreter takes when actually executing a program.

3.4. The Interpreter Loop

When executing a program, the interpreter has to repeat the same set of steps for every instruction. These steps are similar to what a common interpreter does for executing a program, but are extended by a few points because of our special requirements of secure execution.

In general, an interpreter executes a program with a fetch-execute cycle, similar to what a CPU does when executing code. 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 3.3 lists the individual steps taken during the interpreter loop in a textual manner. Figure 3.4 shows a flowchart version of the process. As you can see, 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 to step 1.

Figure 3.3.: Individual steps the interpreter repeats during a cycle.

A cycle consists of multiple instructions executed in succession. 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. During the cycle, the row registers hold decrypted parts of the program, and these registers are principally free to access for any process. As described in the previous section, also the AES round keys are stored in the AVX registers. The protection of those is even more important. If the round keys are leaked, an adversary can easily reverse generate the cipher keys from them (in fact, for AES-256, the first two round keys even *match* the cipher key). Then, not only the interpreted program's encryption is negated, but this also removes protection from any data encrypted with the TRESOR system. Therefore, it must be ensured that no other process copies any register content out to memory, by prohibiting any other process to run during a cycle.

Because the interpreter runs in kernel mode, it has access to kernel functions which can create a sufficient atomic section. Preemptive kernel scheduling can be disabled with the function `preempt_disable`. This prevents the kernel from interrupting the interpreter thread and scheduling in another process instead. However, another function has to be called to provide atomicity: `local_irq_save`. A call of this function saves the current interrupt state and disables interrupts, which is necessary because interrupt handlers may also write the register state to memory. Note that `local_irq_save` disables interrupts only on the CPU it was called from, so on systems with multiple CPUs, interrupts (e.g. from a key press on the keyboard) can be processed without compromising interactivity. To end the atomic section after an interpreter cycle has ended, one calls the reverse functions: `local_irq_restore`, which reinserts the previously saved interrupt state and enables interrupts, and `preempt_enable`, which allows kernel scheduling again.

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. These are needed multiple times during one interpreter cycle, because several encryption and decryption operations are taking place, so it would be inefficient to generate them on-the-fly for every single encryption, like TRESOR does. The next step is to load the program state to registers, so that the interpreter instructions can operate properly. This means that for each segment, a row gets decrypted to the corresponding row register, as described in section 3.2. What was not mentioned before, is that, internally, every segment pointer in memory is mapped to a general purpose CPU register, e.g. the instruction pointer to register `r11`. This is done because these pointers have to be accessed very often during program execution. Instead of always accessing them from memory, it is more efficient to hold them in (otherwise unused) registers during the interpreter loop. In the "load program state step", the state pointers get loaded from memory in their respective register. Now, the interpreter is ready to process bytecode instructions. The used technique here is *indirect threading*, and it was already discussed generically in section 2.1. Each bytecode instruction is represented by a single routine in `km/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 an array of function pointers – in a jump table. An opcode is now just an index into this array of pointers, to the function which represents the bytecode instruction corresponding to said opcode. To execute a bytecode instruction, the interpreter extracts the opcode of the current instruction from the instruction row. The interpreter takes the jump table entry at the index specified by the extracted opcode, and uses it as a function pointer to jump to. In this function, the bytecode instruction is then simulated. Which exact steps are taken, differs from bytecode instruction to instruction. Section 3.5 describes the functionality of each bytecode instruction.

After the instruction is finished, the instruction pointer gets updated to point to the instruction to be executed next. It is possible the next instruction does not lie in the currently loaded instruction row. If this is the case, the appropriate row is loaded from the code segment and decrypted to the instruction row register. The interpreter can now extract the opcode of this next instruction and start the loop once more, but before this is done, two things have been checked. First, it is checked if the amount of instructions executed in this cycle exceed the specified maximum amount of instructions in one cycle. If this is true, the cycle is ended. Second, the halt flag status is checked; a set halt flag indicates that the program should be terminated. The halt flag can be set either by the special `finish` instruction, or due to the occurrence of a runtime error, e.g. a division through zero, or a stack overflow. So, on cycle end or if the halt flag is set, the next instruction is not executed.

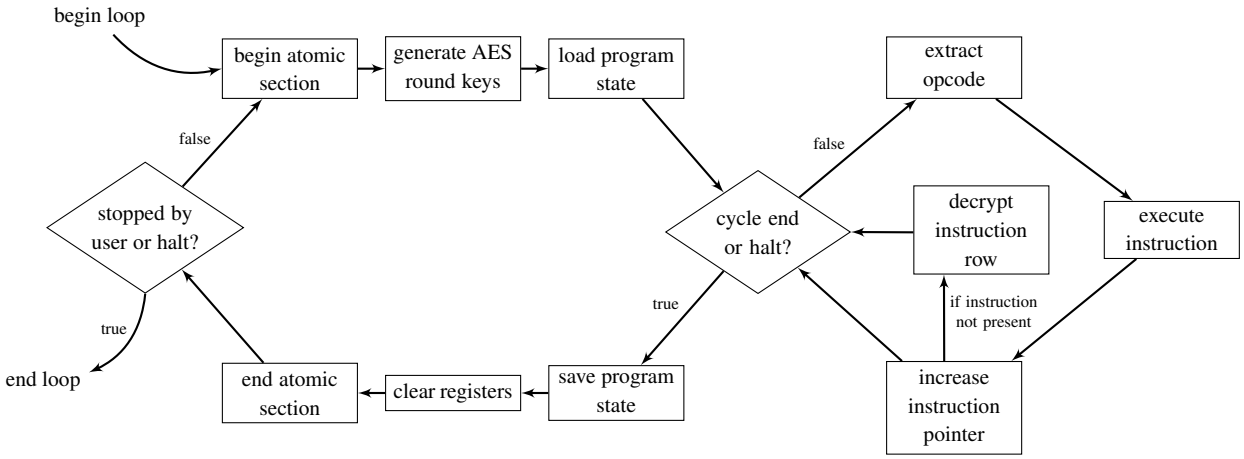


Figure 3.4.: The different steps the interpreter repeats in the interpreter loop. One pass represents an atomic cycle, in which program data and encryption state are protected from leaking to memory.

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. In principal, the next cycle can immediately begin, but before that, the interpreter checks if execution should be stopped. This can either be because the halt flag is set, or because the interpreter thread was issued to stop from the outside, e.g. from the user. In either case, the interpreter loop is broken, and the interpreter thread terminates itself.

One could ask why the splitting of the loop in cycles is needed altogether. It should also be possible to enclose the whole loop in a big atomic section, so that before execution begins, the section is entered, and after execution has finished, the section is left. Indeed, this would protect the registers just fine, and it would even result in performance advantages, because the overhead of loading and saving the state to memory, and generating the round keys each cycle would be avoided. The reasoning against one big atomic section is that it runs for an undetermined time. If, due to a programming error, the interpreted program contains an infinite loop, the kernel thread would run indefinitely in an atomic CPU section, effectively locking up the kernel – which is clearly no good idea. In the cycle model, there are still periods of time where no atomic section is effective, namely between the cycles. During those periods, the kernel can step in and force the interpreter thread to yield, if another process needs to run instead. Also, this gives the guarantee that the interpreter thread will quit if ordered by the user, because the interpreter now has a deterministically reached point to check for termination. The amount of instructions processed in one cycle is configurable per program execution too, which allows the user to compromise between best performance (long cycles, low overhead) and system responsiveness (short cycles, more CPU time for other processes). In section 4.2, the performance of different instruction-per-cycle configurations are benchmarked against each other.

Last in this section, we discuss an issue affecting security that occurred during implementing the interpreter, and how it was solved. Our security policy forbids that any content of the interpreted program touches main memory. We make sure this happens by encrypting any data before placing it in memory. But there are more ways an adversary could gather information about the running program, namely over the interpreter itself. On a x86-64 CPU, the next instruction of an executed program is pointed to by the instruction pointer in register `rip`. Of course, this also holds true for our interpreter. The interpreter has various routines which implement the different bytecode instructions. This means that, during interpretation of a program, the instruction pointer of the interpreter is pointing into these routines. Thus, by observing the instruction pointer of the interpreter, one can indirectly infer which bytecode instructions the interpreted program actually consists of.

Fortunately, the interpreter is already defended against this attack. The routines implementing the bytecode instructions are only executed during the interpreter cycle, and the interpreter cycle is enclosed in an atomic section. This means there is no possible way to observe the instruction pointer of the CPU during the period of time the critical routines are executed.

But there is another small related detail to take care of. When a subroutine gets called, the *return address* gets pushed on the stack, which is the address of the instruction where execution is resumed after the subroutine has ended. Thus, if the interpreter calls a subroutine from one of the bytecode instruction routines, the return address (an address within the bytecode instruction routine!) gets pushed on the stack. As we previously saw, such an address allows deduction of the actual executed bytecode instruction. Even worse, this return address lies on the stack, thus in main memory – which contradicts the security goals. Granted, the leaked information about a single instruction causes nearly no actual damage, but it is desirable to thwart even theoretical attacks.

The chosen solution to this problem is very simple: avoid the calling of subroutines during the interpreter cycle, if possible. But the AES implementing routines from the crypto-module have to be somehow called. An Assembler based work-around for protecting the instruction pointer is shown here:
Instead of the normal method for calling the encrypt routine

```
call bispe_encblk
```

one can use

```
lea 5(%rip),%r9
jmp bispe_encblk
```

which loads the return address (after the `jmp` instruction) into the `%r9` register, and *jumps* to the encrypt routine. So instead of using the stack as storage for the return address, a secure register is used. For returning after encryption is finished

```
jmp *%r9
```

instead of just

```
ret
```

has to be used, which performs a jump to the address stored in the `%r9` register, where the return address was previously placed.
Generally, it might seem unimportant to discuss such a relatively small problem. But this section can act as a reminder that it is very easy to overlook potential security leaks in seemingly secure systems – which can be caused by such small problems.

With this, we conclude this section. We discussed why atomicity is needed during execution, and showed how atomic sections can be the implemented using kernel functions. We followed the interpreter along the different steps taken during the interpreter loop, from the start to termination of the interpreter thread. Last, we discussed how information about the interpreted program can be gathered indirectly by observing the interpreter, and how these issues can be solved.

3.5. Bytecode Language

In this section, we briefly talk about the different bytecode instructions the interpreter supports, and how subroutines are implemented in bytecode. This is followed up with the demonstration of a short, exemplary bytecode program. The section finishes with a discussion of limitations of the bytecode language.

In figure 3.5, all currently implemented bytecode instructions are listed, sorted by opcode. If the instruction has an argument, it is specified in the third column, otherwise, this field is left empty. The fourth column describes how the operand stack changes through the execution of this instruction. The `add` instruction, for example, removes the top two values from the stack, adds them, and puts the result on the stack, so the fourth column for `add` reads "value1, value2 $\rightarrow$ result". If the instruction does not change the operand stack, this field is left empty.

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 than 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)

Figure 3.5.: Listing of all bytecode instructions.

The instructions can be divided into three categories, excluding a few extra instructions which do not really fit in a category. There are instructions for data movement, arithmetic instructions, and instructions for modifying the control flow of the program. Let us begin with the *data movement instructions*.

To load a constant on the stack, one would use the `push` instruction. This instruction takes an integer value as its argument, and places this value on the stack, where other instructions can process it further. Roughly the reverse instruction is `print`, which removes the top element from the stack, and writes it to the output buffer. This output buffer is allocated before execution begins, collects all printed elements, and is copied to the user once execution is finished. This instruction is currently the only way to output data to the user, so it is used to report computation results. To move data between the stack and local variables, the instructions `load` and `store` are used. They push an integer taken from a variable to the stack, respectively pop an integer from the stack to a variable. These instructions expect an index as their argument, which indicates which local variable is accessed. Internally, this index is used as an displacement relative to the call stack pointer. The address where the local variable lies on the call stack, is given by `call_ptr - 4 * index` (the index has to be multiplied by four because it is given in four byte units, not single bytes).

```
int add5(int x) {
    return x + 5;
}

void main(void) {
    print add5(2);
}
```

```
// entry point
00: call 09
02: finish
// int add5(int x):
03: load 1
05: push 5
07: add
08: ret
// void main(void):
09: prolog 1
11: push 2
13: store 0
15: call 03
17: print
18: epilog 1
20: ret
```

Figure 3.6.: Example program in SCLL code and compiled to bytecode.

The next instruction category are the *arithmetic instructions*. Each of those instructions takes the two topmost integers from the stack and performs an arithmetic operation on them. The resulting value is pushed back on the stack. Available are the four basic arithmetical operations in `add`, `sub`, `mul`, `div`, and additionally, the modulo operation `mod`.
The last instruction category are the *control flow modifying instructions*, which can be further divided into jumps and subroutine handling instructions. Each of the jump instructions take an address as their argument, to which the instruction pointer should "jump". This address is not specified relative to the instruction pointer, but absolute to the start address of the code segment. A basic unconditional jump, which always jumps to the target address, is implemented by `jmp`. Then there are a variety of conditional jumps which compare the top two values on the stack by a certain criterion, and then jump to the target address only if the condition applies. Otherwise, the control flow is resumed normal with the instruction after the jump instruction. To get an overview over the different conditional jump instructions, we refer to figure 3.5.

Finally, we describe how subroutines are implemented in bytecode. A program invokes a subroutine with the `call` instruction. This instruction takes the address of the subroutine to call as its argument. When `call` is executed, it pushes the return address on the stack, which is just the address of the following instruction. Then the instruction pointer gets changed to the start address of the subroutine. A subroutine usually starts with the prolog instruction, fittingly called `prolog`, which creates the stack frame of the called function. This instruction allocates space on the call stack, for an amount of local variables specified in the instruction argument. To do so, the call stack pointer gets increased, which creates space for new elements on the stack. Note that the arguments of a function actually belong to the stack frame of the calling function. Thus, any function has to allocate the space for arguments of functions, it plans to call, for itself. A subroutine is ended by the epilog instruction, `epilog`, which frees the previously allocated space by decreasing the call stack pointer again. Finally, to return back to the calling routine, `ret` is used. `ret` pops the return address, previously pushed by `call`, from the stack, and changes the instruction pointer to this address. The function return value is passed over the operand stack: After a subroutine ends, the topmost stack entry would contain the return value, if there is any.

Now, to better illustrate the interactions of the different bytecode instructions, we discuss a short example program. Figure 3.6 shows this program in SCLL code, the language created to simplify programming for the interpreter. On the right is the corresponding bytecode, as generated by the compiler (with comments added for readability). As one can easily grasp from the code, the program calls a subroutine `add5`, which adds the constant `5` to the arguments passed to it. The code in itself should need no further explanation, except the `print` command: This is a built-in keyword which prints the argument to the output buffer, similar to the C function `printf`. It is directly mapped to the `print` bytecode instruction.

The bytecode always starts with the entry point, which the compiler adds in front. This is simply a call to the main function at address `09`, followed by a `finish` instruction, so that the program terminates once the main function returns. At address `09`, the main function begins with the prolog, where space for one variable is allocated on the call stack. Then, the subroutine argument `2` is stored in this variable (at index `0`), because this is the variable in which the subroutine argument is passed. After that, the main function calls `add5`, at instruction `03`.

Because `add5` has neither local variables nor calls a function, it does not need to allocate space on the call stack, so the function prolog is omitted. To push the argument `x` to the operand stack, the variable at index `1` is loaded. It is indexed with `1`, although it is the only variable used in this function; this is because on the call stack at index `0` lies the return address, and only after that the function arguments. Then the constant `5` is pushed on the operand stack, and added to the argument. `add` leaves the result on the stack, which, after returning to `main`, the `print` instruction picks up and outputs to the user. The main function finishes with the function epilog, which releases the allocated stack space, and returns to the entry point, where execution is terminated by `finish`.

To finish up this section, we also have to talk about the limitations of the bytecode language. There are a few prominent features missing. First, a program has no global state, so there is no possibility to allocate static (global) variables. This prevents routines having an easy way to share data. The common alternative to sharing data globally between functions is to pass the data by reference, with a pointer. This, too, is not possible, because the concept of a "pointer" to a variable is unknown to the language. Another absent functionality is the one of an array data type. Both of them are essential to any programming language. And in general, there is only one supported data type; there is no possibility to define composite data types, like with the `struct`-keyword in the C-language. All in all, this severe lack of features currently prevents this language to be used as a production language, like C, or Java. However, this is also not at all the ambition of this interpreter concept. Besides, these missing features of the bytecode language should not be attributed to inherent limitations of the shown interpreter concept, but rather to the limited time frame of this thesis. In section 4.1, we discuss how a few currently missing features could be incorporated into the interpreter in the future.

4 EVALUATION

In this chapter, we evaluate the developed interpreter concept and implementation in regards to several criteria. In section 4.1, we investigate how compatible the interpreter is with existing hardware, software and programming languages. In section 4.2, we discuss several benchmarks comparing the performance of the interpreter against other programming languages. The short section 4.3 provides an argument that the utilized encryption works correctly. In the last section, 4.4, we deliver an analysis of the interpreter's security against memory-, software-, and hardware-attacks.

4.1. Compatibility

In this section we want to examine how compatible our interpreter is with existing hardware and software. We also make some considerations about the degree the C language is already compilable to the bytecode language, and propose how the interpreter could be extended by two more features of C.

First, the interpreter is only compatible with the x86-64 architecture, because on 32 bit architectures, only eight AVX registers are available [34], which, considering the interpreter's design, is not sufficient. On the hardware side, the interpreter requires two CPU features: AES-NI [29], because of its use in the encryption algorithm, and the AVX [15], because the additional register space is necessary to manage program- and encryption-state. This restricts usage to the x86 architecture, as these instruction sets are currently not available on any other processor architecture, like, for example, the ARM architecture. AES-NI is offered by both Intel and AMD processors. The first Intel microarchitecture to support AES-NI was Westmere in Q1, 2010. Since then, the following architectures Sandy Bridge, Ivy Bridge, Haswell and the newest release Broadwell all supported AES-NI. AMD introduced AES-NI with the Bulldozer platform in Q4, 2011. All subsequently released AMD platforms also provide the instruction set. AVX was first introduced by Intel with the Sandy Bridge architecture in Q1, 2011. All the above named newer architectures from Intel support AVX as well. AMD offered AVX the first time with the above mentioned Bulldozer platform. For AMD, also all further released platforms support the extension.

Both AES-NI and AVX are not available in every single CPU of the above listed architectures. The lower priced CPUs, often from the mobile and embedded segments, do not necessarily support the extensions. But for every new microarchitecture released, there is an increasing percentage of CPUs per architecture supporting the extensions. It can be expected that in the future, the spread of CPUs supporting AES-NI and AVX will further increase, at least on the x86 platform.

On the software side, the interpreter is currently only implemented as a Linux kernel module, which restricts deployment to the Linux operating system. As the implementation is based on the TRESOR kernel patch, the range is further narrowed down to the TRESOR supported kernel versions, which currently are 3.6.2 and 3.8.2. However, TRESOR is not inherently limited to these kernel versions, but is compatible with all newer kernel versions. The same is true for our interpreter – in respect to TRESOR, it adds no kernel support breaking changes. On a side note, it is also possible to run the interpreter stand-alone, without the TRESOR patch. This was successfully tested with kernel version 3.13. We do not recommend this option though, because without the TRESOR patch, significant security weaknesses are introduced. For example, TRESOR blocks the accessing of debug registers for user space applications, and without this, any program could just read out the used cipher key. Additionally, there is no longer a good, secure way of acquiring the password from the user, which TRESOR does pre-boot.
To mention it, AVX also requires support by the operating system. But this does not present a problem, as Linux provides AVX since kernel version 2.6.30 [40]. Microsoft Windows offers AVX since version 7 SP1 [41] – a Windows implementation of the interpreter is also an interesting possibility. There should be no fundamental obstacles preventing a Windows portation, but this path was not yet explored at all.

There are a few limitations that the interpreter has inherited from TRESOR. Those should also be mentioned here. Because the debug registers are reserved for storing the cipher key, any application that depends on using the debug registers for their original purpose, namely holding break points, will no longer work. This mainly affects debugging software such as GDB. Fortunately, break points can be emulated well in software, and GDB even standardly uses those. This means that for most users, which do not have an urgent need for hardware break points, usage experience is not limited. A further restriction is that the interpreter can not be run in a virtual machine, at least not securely. A VM simulates the guest system within the host system, which means that the VM's CPU registers are stored in the host system's memory. Thus the cipher key used in the VM is exposed to a cold boot attack on the host system's memory. Therefore, the interpreter can only be run on real hardware, and not within virtual machines.

Another subject to investigate is how compatible the bytecode language, respectively the interpreter, is to existing programming languages. It would be desirable to have a compiler that is able to take, say, C source code, and generate interpreter bytecode from that. This would allow an easy way to employ the interpreter in practice. The foundation for that was already created with the compiler that compiles "secure C-like language" code to bytecode. The secure C-like language is pretty much a subset of the standard C language (sans the `print` statement), as can be reproduced by looking at the language grammar in section A.1. So we can say that in its present form, the interpreter is compatible to at least parts of the C language. Of course, as outlined in section 3.5, quite a few missing features would have to be implemented before it would be fully possible to compile C code to bytecode. To illustrate that the interpreter is not limited to its current feature state, we will sketch how two currently absent features could be integrated into the interpreter: arrays and static variables.

An array is just a series of variables lying consecutively in memory. Like variables, arrays would be allocated on the call stack, in the stack frame of the function it is contained in. The compiler, which knows the size of the array, assigns the array an address, and reserves the fitting amount of space in the function frame. So allocating arrays is a means of the compiler, with the tools already in the language. Accessing array elements requires extending the instruction set. Each array element is identified by the starting address of the array, and an index representing the element position in the array. With this in mind, one can easily design two new bytecode instructions which transfer elements between an array and the operand stack, named for example `aload` and `astore`. These would work similar to the existing instructions `load` and `store`, but in addition to an address, these instructions would require the index of the element to access on the operand stack. The instructions can then calculate the address of the target element from the base address of the array, and the element index. After having the address of the target element, the instructions can be implemented just the same as `load` and `store`.

Static (global) variables can not be simply implemented by introducing new instructions. These variables should be accessible over the entire runtime of the program, and also from everywhere in the program. As there is currently no segment, where static variables could be stored, that satisfies these properties, a new, "global" memory segment would have to be introduced. Is that possible? Like the other memory segments, the global segment would need its own row register, which contains the decrypted slice of the segment, and a pointer to the currently loaded global element. Neither constraint poses a problem. The currently unused registers `XMM7` and `r15` could carry the global segment row, respectively the pointer to the currently loaded element. The interpreter would allocate this segment together with the other segments, before program execution is started. Before the main function is called, the interpreter initializes all static variables with their preassigned values.
New bytecode instructions would also have to be introduced for transferring variables between the global segment and the operand stack, named for example `gload` and `gstore`. The implementation of those would also be pretty similar to the `load` and `store` instructions. Extra thoughts would have to be spent on how the segment is encrypted. Because the global segment should be random accessible, like the call stack segment, the encryption in CBC mode is expensive, as a change to an element triggers a re-encrypt up to the end of the segment (this effect was already described in section 3.3). The bigger the segment becomes, the more time a write to the segment consumes, with possibly hefty impacts on performance. To put a stop to that, the segment could be divided into chunks, which get encrypted independently of each other, so that a write never triggers more than a certain amount of encryption operations. Aside from performance considerations, it should be possible to introduce a global segment to the interpreter, resulting in the ability to use static variables.

To sum up, the interpreter is supported by most x86 processors released in the last few years. The interpreter runs only on Linux, but in itself has no special requirements to the kernel version, so it should be compatible with all kernel versions TRESOR can be ported to. Theoretically, the should be no fundamental restrictions preventing a portation to other operating systems. The bytecode language is already compatible with a part of the C language, and we proposed how two missing features could be incorporated into the language. It is unknown if it is possible to implement *all* features needed to make the interpreter fully compatible to C. Considering the two examples portrayed above, the author is mildly optimistic that this might be achievable.

4.2. Performance

In this section we will investigate the performance of the interpreter. Given the interpreter's design, a performance drop-off in comparison to other execution environments is expected. The interpreter is forced to decrypt code and data before it can be processed, whereas other programming languages do not have this disadvantage. The use of the AVX registers often requires to use awkward solutions that could be solved more straight forward in normal interpreters. 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 of course 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 was Xubuntu Linux 14.04, with kernel version 3.8.2, TRESOR patch applied.

The remaining three languages were selected to fit into different kinds of execution types. The first choice is the C language. 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 following flags were used:

```
-std=99 -Wall -Werror
```

Note that we did chose not to turn on GCC's optimization features. The reasoning behind that is that our own language's compiler does not implement any optimizations whatsoever. As we do not want to benchmark compiler sophistication here, and to enable a fairer comparison, optimization was left off. The second language chosen is the Java language. Java represents the class of JIT-compiled interpreted languages. As mentioned before, just-in-time compilation is a technique in which bytecode is compiled to machine code at runtime, which can be executed natively by the CPU again. Java is also compiled from source code to bytecode before execution, which spares the interpreter the overhead of parsing the source code at runtime. 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 CPython implementation, in version 2.7.6. Python represents a simpler kind of interpreter implementation. The CPython interpreter is implemented as a slower type of interpreter. CPython uses no JIT compilation, and the source code is parsed to bytecode just before execution. This should make Python's performance results closest to our interpreter's performance. Concerning our interpreter, the programs were run with the chosen default value, 2000 instructions per cycle.

Creating the programs to benchmark presented us with the problem that our bytecode language is limited in its features. This restricted the types of problems we could implement to relatively simple mathematical calculations. We were further constrained by the fact that our interpreter only has a 4 byte integer data type. Most mathematical problems quickly exceed the range of numbers one can represent with 4 bytes, and a 8 byte integer would actually be needed. Because we do not have such a data type at our disposal, we had to limit the input size of our problems during benchmarking. The programs we finally chose to implement calculate the following:

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

These are no sophisticated problems, but they should be sufficient for a basic performance comparison. You can find the implementation of the three programs, in the interpreter's language, in appendix A.2. The C, Java, and Python implementations were ported outgoing from that implementation. As the programs are relatively simple, no special adaptations were necessary, and the ports were done in the most straight forward manner.

The elapsed time of a program run has been measured with the `time` shell command. On Linux systems, there are commonly two version of this command, the GNU version and the built-in command of the Bash shell. We went with the Bash version, as it offers millisecond precision instead of centisecond precision.

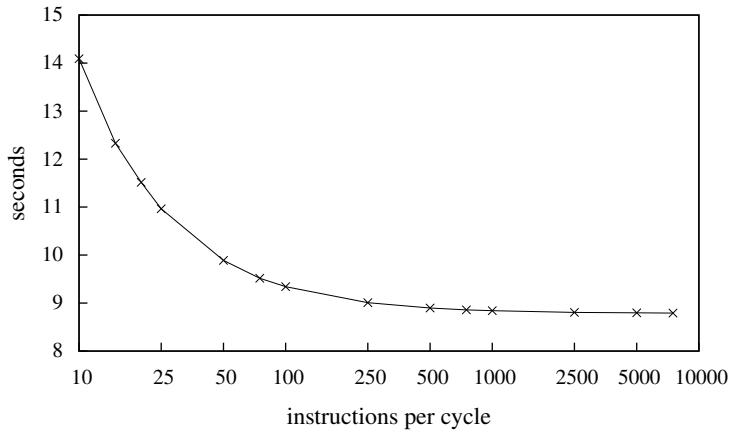


Figure 4.1.: Benchmark of the Fibonacci program to test the influence of different `--instr-per-cycle` settings.

There may be more precise time measurement techniques using options the different languages provide, but first, our interpreter does not offer time measurement, and second, the languages all measure time slightly different, which might make the measurements not comparable. Therefore, we just use `time` as an easy to use, common measurement option.

An additional property of the interpreter is that execution is divided in atomic cycles. Each of these cycles creates a performance overhead. At the beginning of each cycle, the AES round keys must be generated, which is a costly operation. This means that the interpreter execution speed increases when less cycles are used overall during execution. The user can adjust how many cycles are used with the `--instr-per-cycle` option, which controls how many instructions are executed before the cycle is ended. We tested the exact influence of this setting on performance. For this, the Fibonacci program for $n = 35$ was taken and run with different `--instr-per-cycle` settings. The results were averaged out from 20 program runs. You can see the resulting graph in figure 4.1. Low instructions per cycle values have a drastic influence on performance. At 10 instructions per cycle, the cost of generating the round keys makes up almost half of the execution time. With more instructions per cycle, the overhead decreases fast. From 250 onwards, the overhead does no longer play a significant part of execution time, and the differences lie in decisecond range. Between 5000 and 7500 instructions per cycle, the difference amounts to only 4 milliseconds, which is at the border of our measurement accuracy. Drawing from this result, we have chosen the default instructions per cycle value of the interpreter to be 2000.

You can see the benchmark results in figure 4.2, with the results averaged out from 50 program runs. Here, the column labeled "SCLL" contains the benchmark results of our interpreter. Another figure 4.3 shows the calculated performance ratios between the interpreter and the other tested languages. A first overview shows roughly the same picture for all three programs. As expected, the interpreter is quite a bit slower than the next slowest language, which is Python. C and Java perform both much faster than Python and SCLL, in fact, 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. Both simpler interpreter's implementations – our's and Python's – can not hold up with that. If we compare our interpreter to Python, we can see that the interpreter performs reasonably. On average, SCLL is about a factor 4 slower than Python.
The influence of encryption on the interpreter's performance is very interesting, and we decided to measure that as well. No complicated adaptations were necessary for that, as the interpreter kernel module offers deactivation of encryption per compile-switch. In figure 4.2 the results of this benchmark are shown as striped bars within the SCLL bar. Here, it is easy to see that encryption takes up a major part of the interpreter's runtime. Without encryption, the interpreter was between 3 and 6 times faster than with it. On average, the interpreter spends four fifth of the overall runtime with encryption. Actually, the interpreter's performance without encryption is on par with Python's, in case of the Fibonacci program, SCLL is even 40 % faster than Python.

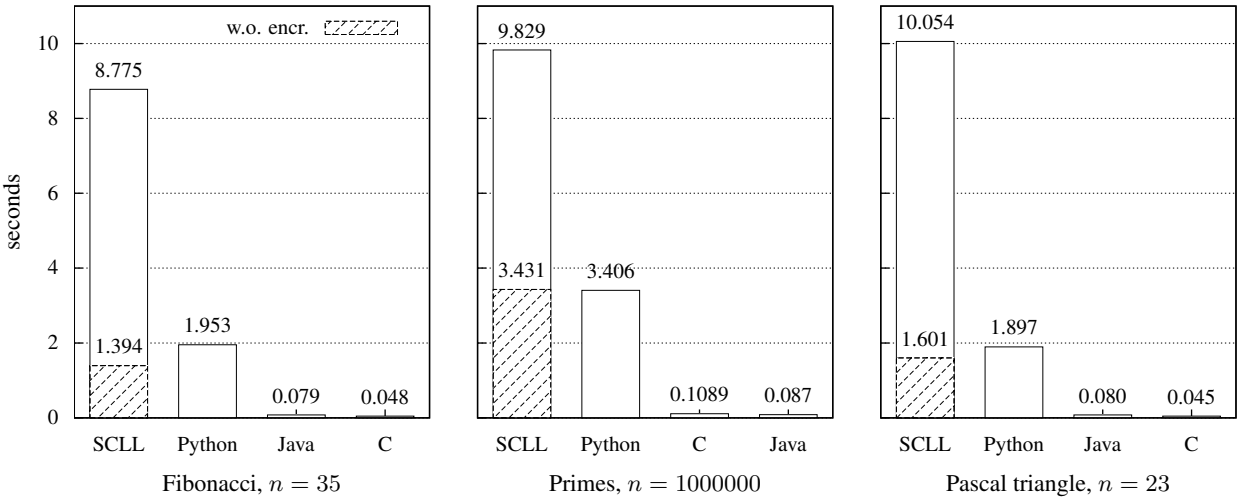


Figure 4.2.: The results of the language benchmark show that the interpreter is vastly slower than C and Java, but still within performance reach of Python.

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 behind that, we have to take a look at the source codes of the programs. Here we can see that Primes is implemented purely iterative, Fibonacci purely recursive, and Pascal uses both iteration and recursion. This indicates that recursive programs affect the interpreter's performance negatively. Recursion requires the repetitive use function calls. Within each function call, a stack frame has to be allocated and freed which creates encryption overhead at the interpreter. This also fits if we take the benchmark of the encryption-less interpreter in account. 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-sixth* encryption time. Further benchmarks are necessary to come to a final evaluation in this question, but this is already a first indication that the interpreter is weak at handling recursion.

Summarizing, we have benchmarked the interpreter against three other popular programming languages: C, Java, and Python. Whereas the former two are about 100 to 200 times faster than our interpreter, the latter stays in performance reach, being about four times faster. 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's. One has to say that the benchmarked programs we used here really qualify only for microbenchmarks. Currently, performance tests with more meaningful, real-world programs are not possible due to the restricted feature set of the interpreter. Keep in mind though that the interpreter's ultimate goal is to provide security. Given that the AES computations are included in our time measurements, a negative impact on performance is to some degree unavoidable and future work will focus on reducing this.

	Python	Java	C	SCLL, w.o. enc
Fibonacci	4.5	111.0	182.8	6.3
Primes	2.9	113.0	90.0	2.9
Pascal	5.3	125.7	223.4	6.3
Average	4.2	116.3	165.4	5.2

Figure 4.3.: Performance ratios between SCLL and the other benchmarked languages. Java and C are between one and two magnitudes faster than the interpreter, Python about 4 times. With encryption turned off, the interpreter is about 5 times faster.

4.3. Correctness of AES Implementation

The foundation to a secure implementation is of course that the encryption works correctly. In this section, we shortly show that this is the case for the employed AES encryption. The interpreter uses the AES algorithm adapted from TRESOR. In the TRESOR paper [13], it was already shown that their AES algorithm works correctly. The only modifications to the algorithm were changing source and destination registers the instructions work with. The instructions themselves stayed untouched, so the algorithm should still work correctly, as before. However, to check that no error somehow slipped into the implementation, the algorithm was tested again.

The Linux kernel's cryptographic API allows the registering of new encryption algorithms, so that applications can use them in a structured way. We do not need anyone else to use our specialized algorithm, but the crypto API also offers a test manager, which checks registered algorithms on correctness with test vectors. If the algorithm does not work correctly, the results differ from the known test vector results. To make use of that, a small kernel module, `bispe_crypto_test`, has been written, which registers the interpreter's AES implementation with the crypto API. Then it instructs the test manager to check the algorithm by applying the official AES test vectors [16]. As expected, the checks of the test manager run through without failure. Based on the tests already done in the TRESOR work, and our additional tests with help of the crypto API, it can be argued that the interpreters AES encryption behaves correctly.

4.4. Security

The most important property of the interpreter system is of course the security it can provide. In section 4.4.1, 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 section 4.4.2, and investigate possible weaknesses of the interpreter against hardware attacks in section 4.4.3.

4.4.1. Protection Against Memory Attacks

The main purpose of the interpreter is to protect programs against memory attacks, such as the cold boot attack. In order to achieve that, we set the security goal for the interpreter to not using main memory for any sensitive data. This means that data can only reside in memory encrypted, and, in reverse, that it can show up decrypted only in secure storage. We also set ourselves the goal of not weakening the given security of a system provided by TRESOR. This means that changes introduced by the interpreter may neither leak the cipher key nor the derived AES round keys from the secure storage where TRESOR put it. A leaked cipher key does not only break TRESOR's disk encryption, but also removes all efforts spent on protecting the interpreted program by encryption, so this second goal is also directly important for the interpreter's security. The two goals are closely related, and thus, we check their adherence together in this section.

In TRESOR's case, the secure storage, where the cipher key is put, are the CPU debug registers, and, during encryption, the AVX registers for the AES round keys. The interpreter expands on that and additionally holds the "segment rows" in the AVX registers, which are decrypted parts of the interpreted programs code and data. So, to satisfy the security goals, it must be ensured that parts of neither the debug registers nor the AVX registers get somehow leaked to memory. We have to consider the circumstances under which such leaks are possible. The interpreter in itself is implemented in a way that decrypted parts of the program are only held in registers, never in in memory. The encryption algorithm is also implemented to refrain from using memory, as this was the original goal of TRESOR. Thus, theoretically there should be no data leaked to memory. A more detailed verification in practice is desirable nonetheless.

But there are other ways through which the registers could be leaked to main memory. An obvious way is the OS, an application or an attacker just reading out the debug registers, in which the cipher key is stored. The TRESOR authors anticipated that and patched the kernel code, through which the debug registers are accessed, to no longer give away the register's content. As we are operating under the prerequisite that

the TRESOR kernel patch is used, the interpreter is also protected against this attack vector. Another path of leakage is opened through the operation system (OS). When the OS switches between two processes, it saves the state of the old process in memory, so that this state can be restored again later when the process is resumed. The state is known as the process's context, and it normally includes the debug registers as well as the AVX registers. Debug register access is patched, so these are no longer included in the saved process context, but AVX registers still are. Therefore, during interpreter runtime, every time the OS schedules out the interpreter process or an interrupt occurs, sensitive data is possibly leaked. TRESOR solves this problem by running the encryption of one block atomically, which means that the operating system can not interfere with context switches in between. To protect it against context switches as well, this solution was adapted in the interpreter implementation. As described in section 3.4, interpreter cycles are enclosed in atomic CPU sections. The atomic section spans around the time where sensitive data is contained in the AVX registers, and before the section is left, all registers get cleared before anyone else can have access again. Therefore, at least by design, no AVX register containing interpreter data or the AES round keys should slip through to RAM.

The TRESOR authors already performed extensive tests to verify that their system is protected against memory leaks. This was done by scanning the main memory of a patched system for the cipher key and also parts of the cipher key. The results support the claim that the used approach protects against memory leaks: no parts of the cipher key longer than three bytes could be found. According to the authors, these three byte findings can be explained statistically, and they also occurred on a scanned system without TRESOR patch. As the interpreter uses the same protection approach as TRESOR, we can lean on these results and thus be already confident that the interpreter satisfies our overall security goal. Of course, for our security analysis, we have not solely relied on their tests, because our implementation uses a slightly modified version of TRESOR's AES and the interpreter accesses memory as well.

We therefore performed our own memory scans of a system running the interpreter. To obtain dumps of physical memory, we chose to use the same approach the TRESOR authors use, namely using a Qemu/KVM virtual machine [42, 43]. A Debian Linux with kernel version 3.8.2, TRESOR patched, was installed into the virtual machine. Through the debug console, Qemu allows to take complete snapshots of the physical memory of the guest system, outgoing from the host system. For analyzing the memory dump, a little program, `mem_check`, was written. This program takes multiple 16 byte patterns as input, and scans the memory dump for the longest matching byte sequence of each pattern, in little and big endian.
There are multiple patterns that should not occur in the memory dump. The first investigation was to test if running the interpreter somehow leaks the cipher key or the AES round keys to memory. To check this, we generated the full AES key schedule for a known password, and used every round key (which also includes the cipher key) as 16 byte search patterns in our scan program. In preparation of the memory dump, the virtual machine with TRESOR patch was booted, and TRESOR was given the same, known password to use as the cipher key. A few memory dumps were taken at different times. The first memory dump was taken without running the interpreter at all, to compare if running the interpreter influenced the scan results altogether. Then 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.

You can see the condensed results of the memory scans in figure 4.4. Additionally, all search patterns and the corresponding scan results are listed in appendix A.3. The results of the memory scan matched the observations made by the TRESOR test. None of the 16 byte round keys was found completely in memory. The longest matching byte sequence for almost all the round keys was three bytes. Only for two round keys in big endian, a four byte sequence was found, but that should also be attributed to coincidence – in memory, the round keys are most likely stored as little endian anyways. In fact, the exact same scan result was found for all different memory dumps taken. This indicates that through running the interpreter, the round keys are not leaked to memory. Which asserts our presumption that by using the same protection approach as TRESOR, the interpreter is protected from memory leaks too.

Memory dump taken	without interpreter	during Fib run	after 100 Fib runs	during Random program	after 1000 Random runs
AES round keys	3 / 3.07 / 4	3 / 3.07 / 4	3 / 3.07 / 4	3 / 3.07 / 4	3 / 3.07 / 4
Fib program, code	11 / 12.47 / 16	11 / 12.47 / 16	11 / 12.47 / 16	–	–
Random program, code	5 / 5.31 / 6	–	–	5 / 5.31 / 6	5 / 5.31 / 6
Random program, data	3 / 3 / 3	–	–	3 / 3 / 3	3 / 3.06 / 4

Figure 4.4.: Shortest/average/longest byte sequence match for different 16 byte search patters and memory dumps.

In accordance with our first security goal, we also wanted to know if we can find any memory traces of the code and data of a program executed by the interpreter. In principle, code and data should show up decrypted only in AVX registers. As the round keys are also stored only in AVX registers, the previously found results should also apply for an interpreter program's code and data. Nevertheless, the taken memory dumps were scanned for further patterns. As search patterns, the decrypted bytecode of the Fibonacci program was taken, cut into 16 byte fractions, like the interpreter would load them decrypted into the segment row registers. At first to our surprise, the longest matching byte sequences were almost all over eleven bytes long, in some cases even the exact 16 byte pattern of the bytecode fraction was found in memory. However, the reason for this lies most probably in the structure of the bytecode, and not in the interpreter leaking bytecode to memory. One bytecode instructions is 4 bytes long, but the actual opcode can be encoded in a single byte. So, a bytecode instruction consists of three zero bytes, and only one non-zero byte. About 90 % of the memory dump consisted of zero bytes, which is supposedly because a major part of the memory has stayed unused during the time we conducted our scan. Thus, the probability that parts of the bytecode sequences, which also contain mostly zero bytes, show up randomly in the memory dump, is heavily increased, which explains these scan results.

If a program's bytecode contains fewer zero bytes, the byte sequence matches should also be shorter, if we scan the memory for those bytecode patterns. To test that assumption, an interpreter program, that adds 32 randomly generated integers, was created. You can find this program's code in listing A.5, and in figure 4.4, containing the scan results, this program is referenced to as "Random". The idea behind the program is that in the program's bytecode, the `push` instructions that load the 32 random integers on the stack, are less dispersed with zero bytes. Thus, if we use these instructions as search patterns, we can expect shorter byte sequence matches, that is, *if* the interpreter does not leak the bytecode to memory. We can also search for a second type of pattern: While running the program, the interpreter successively loads the 32 random integers to its stack segment. If the interpreter leaks parts of the stack segment to memory, we could be able to find byte sequences of the 32 integers in the memory dump. Thus, we used all the integers as a search pattern, separated into 16 byte fractions, as the interpreter would load them decrypted to the stack row register. Again, one memory dump was taken during the interpreter running the Random program, and one after running the Random program a thousand times.

The scan results matched the expectations: the longest match for the bytecode patterns of the Random program was six bytes long. Those six byte matches are still longer than the three byte matches which occurred "randomly" at the search for AES round keys, but due to the bytecode structure, the search patterns still contain a few zero bytes, which skews the result towards longer byte sequence matches. Drawing from that result, we can confirm that the 16 byte matches of the Fibonacci bytecode search patterns were most likely based on the frequent occurrence of zero bytes, and not on the interpreter leaking to memory. The stack segment search pattern had a similar result in the scan for the round keys. Because this search pattern contained only random data, namely the 32 random integers, and no zero bytes, most of the patterns had a longest match of three bytes; only in one case, the longest found match was four bytes long. This fits to the result of the search for the round keys, which have a similar random byte structure and also longest matches of four bytes. The scan result stayed nearly the same over the different memory dumps. Again, this indicates that running the interpreter has little influence on memory, at least in regard to interpreter data leakage.

To summarize, neither the round keys, nor an executed program's code or data could be fully found in the scanned memory dumps. The byte sequence matches that were found are statistically expectable. Our scan results generally matched the scans conducted by the TRESOR authors, which ought to be expected, as the same protection method in form of atomic sections and the TRESOR kernel patch are used. The results indicate that neither debug register nor AVX registers get leaked to memory, which confirms the adherence of the interpreter's security goal – no sensitive data in memory. However, to obtain perfect affirmation of our security goal we would have had to perform memory dumps and scans for every possible interpreter state, which is hardly feasible. But combining our observations with those of TRESOR, we can say that generally, the interpreter keeps the cipher key, the round keys, and the segment row registers away from memory. If, in the future, some interpreter or even kernel code is still found that violates this rule, the leaking code can most likely be patched.

Concluding, we can state that the interpreter protects executed programs against memory attacks. Additionally, because the interpreter does not leak cipher- or round keys to memory, the interpreter can be run alongside TRESOR, without compromising TRESOR's security against cold-boot attacks.

4.4.2. Protection Against Software Attacks

Another interesting topic is how much the interpreter can protect executed programs against attacks on the software level. With that, an attacker is meant that has access to the same system the interpreter is running on, and wants to obtain code or data of the executed program. This can for example be the case if the system is infected by a Trojan horse trying to spy on sensitive data. A variation of this scenario occurs in cloud computing, where services provide remote computation resources to clients. This remote resource is an unsafe environment; the client can not be sure that his programs are executed unobserved by the service provider or any third party that might have access to the resource. Our bytecode interpreter could offer protection by letting clients execute their programs *encrypted* on the remote server, keeping anyone from monitoring the program. Another scenario are mobile devices, where the user wants some applications to keep their data safe from spying higher privileged applications, or even the operating system. As future work, one can imagine a port of the concept to mobile platforms, such as Android, where the interpreter could find applications in such scenarios. In this section, we discuss some attack scenarios, and how much security the interpreter provides in them.

There are a few imaginable approaches an attacker could try to break the protection of the interpreter, thus gaining access to the code or data of a bytecode program. A trivial first attempt is a direct attack on the encrypted program executable. If the attacker can reverse the AES encryption, the program's code lies open to him. This attack vector is pretty much unusable, as AES is yet considered unbroken in practice. The best known attacks on AES are only of theoretical nature [28]. Another 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, he can create a complete picture of the program's code, and the data the program is working with. However, this would require outside access to the row registers at the time they contain decrypted data. But because the row registers hold decrypted data only within the atomic CPU section, this is not possible. The atomic section prevents any other process that could possibly 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. Before ending the atomic section, the interpreter clears sensitive data from the registers, before any other process can access them again. We also know from the previous section that the interpreter does not leak data to main memory, so the data can also not be accessed by detouring over memory. So this attack can be ruled out as well.

The last avenue of attack, we want to consider here, is relatively effective, but can be defended against with certain measures. If the attacker can get a hold of the used encryption key, all protection provided by the encryption is reverted. The cipher key is stored in the CPU debug registers at all times, unlike the decrypted data, which populates the registers only during the interpreters atomic section. This makes the cipher key difficult to protect, as in principal, anyone can access the debug registers. We therefore have to investigate under which circumstances an attacker can access the debug registers, and thereby read out the used cipher

key. Debug registers are a privileged resource, which means that only ring 0, the kernel, can access them. The debug registers are accessible for user space applications only through the `ptrace` system call. As mentioned previously, TRESOR patches certain kernel code to make the debug register inaccessible – the `ptrace` system call is patched to return `0` instead of the real debug register content. This means that it is impossible for a non-privileged attacker to read out the debug registers. An attacker with root privileges has more possibilities. By using a loadable kernel module (LKM), or `/dev/kmem`, he can insert arbitrary code into the kernel and execute it from within ring 0. TRESOR also patches the standard routines to access the debug registers from within the kernel, but the attacker is not dependent on those routines. His inserted code can just directly read out the registers. In its current form, the interpreter is vulnerable to an attack of this kind. To protect against this security flaw, the TRESOR authors advice 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 lets him execute his code, to still capture the cipher keys. All in all, this would make the interpreter pretty resistant against attacks on the software level.

Another desirable property is that even the kernel itself can not observe the executed program. This is very relevant in the above mentioned cloud scenario, where the remote system, which runs the code, and thus the remote kernel, is untrusted by the owner of the computation. If the executed program is guaranteed to be inaccessible even from the remote kernel, the owner of the computation is no longer dependent on trusting the cloud service provider. Two big problems arise here: First, the kernel can always just read out the used cipher key from the debug registers. For this scenario, the debug registers are no longer suitable for cipher key storage. The second problem is that the integrity of the interpreter itself is not guaranteed. The kernel can patch the code of the running interpreter process in any way to modify its behavior. For example, the usage of atomic sections could be disabled, which in turn allows to read out decrypted data from the registers. These problems currently prohibit the usage of the interpreter in said scenario, and most likely, a hardware solution is required to solve them. In section 5.2, we sketch how a solution could look like.

To summarize, the primary road of attack is to target the cipher key stored in the debug registers. The interpreter is vulnerable against attackers with root privileges, because they can insert malicious code into the kernel, yielding them access to the debug registers. By disabling LKMs and `/dev/kmem`, the interpreter offers protection even against attackers with root privileges. The interpreter's security guarantees against attack on the software level are thus based on the kernel's integrity – in an untrusted environment, like in a cloud service, the interpreter can currently not be run securely.

4.4.3. Protection Against Hardware Attacks

A really simple hardware attack would work if the CPU registers keep their content after rebooting of the system. Then an attacker could simply reboot the system and read out the debug registers with the cipher key, which would destroy the feasibility of the whole concept. Fortunately, according to the TRESOR authors, this is not the case. The CPU registers lose their content instantly if power is cut, which prevents this simple attack vector.

In our discussion of the interpreter's resistance against attacks on memory, we mostly had the cold boot scenario in mind, that is, an attacker has the possibility to read out the RAM content. At the beginning, DMA attacks on memory, such as Firewire attacks [8, 9], were also mentioned briefly. These work in the same vain as cold boot attacks, namely scanning the target system's memory for cipher keys. The interpreter is not vulnerable to those kind of attacks, because it keeps cipher key and program content outside of RAM. More possibilities arise for an attacker that can not only read from RAM, but also *write* to it, and DMA allows exactly that. Blass and Robertson [44] 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. Because this attack works decisively different than the afore mentioned, "read-only" memory attacks, it is included under hardware attacks. The attack works by first connecting an attacker

controlled system to the running target system through a DMA port, e.g. Firewire. Then, the attacker copies an image of the target's physical memory with the purpose of identifying certain kernel structures. With the knowledge how the target's kernel is build up, the attacker crafts a payload which is then injected into the target's memory. Through patching the interrupt descriptor table, the kernel is issued to execute the payload, which is essentially a piece of code that dumps the debug registers containing the cipher keys to memory. After the cipher key is in memory, the attacker can obtain it by reading with DMA.
As this attack works against TRESOR, it will also work against our interpreter. Currently, the interpreter is not protected against this kind of attack, and it is unknown what the best possible defense approach is. The TRESOR-Hunt authors propose a few solutions themselves, such as 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. The easiest solution is the complete deactivation of DMA, which has the drawback that DMA can no longer be used for its performance benefit.

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. Intel's modern CPUs also expose JTAG ports on their surface. JTAG, among other things, allows to read and set CPU registers, thus the debug registers as well. Therefore, in theory, it might be possible to obtain the cipher key through JTAG debugging the CPU. However, we are not aware of anyone successfully carrying out said attack.

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 compatibility, performance and security. We summarize the found limitations in section 5.1. In section 5.2, we present future tasks and investigations that can be pursued to further extend the interpreter's scope of applicability. Finally, in section 5.3, we summarize the work done in this thesis and draw a conclusion about the overall applicability of the interpreter's concept.

5.1. Limitations

Currently, the developed bytecode language supports only a very narrow set of features, which makes it impossible to use in real world applications. To become deployable in practice, a lot of further extensions will have to be made regarding the range of the interpreter's features. However, we have not yet found any obstacles caused by the interpreter's design which will impede future integration of common programming language's features, such as arrays or static variables.

Furthermore, the interpreter's performance is negatively affected by multiple factors. In its current form, the interpreter can not really keep up with the execution speed of other languages. The prime reason behind the interpreter falling behind in performance is caused by its secure design. The interpreter uses encryption for program code and data and is thus inherently at a disadvantage against languages using no encryption. Encryption is integral to the concept, and thus can not be dropped for a performance boost. Related to that, the performance is also affected by the use of the AVX registers, in which we store the decrypted data. This forces us to transfer data between the AVX registers and the general purpose registers before they can be processed. Future AVX instruction set extensions may introduce more possibilities to work directly on the AVX registers, allowing more efficient code. Last, some of the performance losses might also be rooted in the implementation itself. Compared to Java's virtual machine, or Python's interpreter, our interpreter is of course not nearly as mature and optimized. Considering the scope of this thesis, this is to be expected, especially as the primary goal was to develop a *secure* interpreter.

As mentioned before, Java, and other interpreted languages make use of JIT compilation to boost their execution speed. Unfortunately, this technique is not compatible with the interpreter. With JIT, bytecode is translated into native instructions, which the CPU then executes *from memory*. In the interpreter concept, code may only appear decrypted in CPU *registers*, and the CPU can execute instructions neither from a register, nor encrypted from memory. Therefore, JIT compilation can not be utilized by the interpreter.

Security-wise, the interpreter fulfills its original purpose, to protect programs against traditional memory attacks, like cold boot attacks. The interpreter is vulnerable against attacks which exploit the ability of DMA to write to arbitrary memory locations, like the presented TRESOR-Hunt. Attacks of this kind can be averted by disabling DMA, which is no optimal solution, as DMA also has plenty of practical benefits. On the software level, the security of programs executed by the interpreter depends on the integrity of the kernel. The critical debug registers are defended against attackers with user privileges, through patching of the `ptrace` system call, and against root attackers, if the kernel is compiled without LKMs and `/dev/kmem`. In ring 0, the debug registers can no longer be protected, which means an attacker could break the interpreter's security if he can execute his own kernel code, e.g. with a kernel bug. In turn, this also means that executed programs are not protected against the operating system itself, which renders the interpreter unusable in cloud scenarios.

5.2. Future Work

From here on, there are many possible ways to further advance the bytecode interpreter concept. The herewith presented implementation is really only a starting point. In this section, we present some possible future developments.

A big, but certainly worthwhile goal is to extend the bytecode language to be fully or in large part 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 are compilable 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 can also be increased. Currently, a program can be executed and encrypted only with one single key, and to change 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 could somehow be scrambled with the master key set by TRESOR.

In regards to performance, there surely are some instances were optimizations within the current interpreter implementation can be conducted, but no major leaps in performance can be expected from those. This stems from the fact that most computation time is spent on encryption, which is necessary and can hardly be further accelerated, considering that it already runs in hardware. Bigger advances in performance can most likely not be gained until the launch of Intel's AVX-512 instruction set [45]. 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. For the interpreter, it is conceivable that this additional space can be used for *caching*. Currently, when a new segment row is loaded to a register, the old register content has to be encrypted back to memory. With more registers, the old register content can stay cached within the registers, which allows to immediately use it again if needed, without previous decryption from memory. This reduces the overall amount of decryption and encryption processes needed, which should result in notable performance improvements. AVX-512 also brings many new assembler instructions, which may allow simplifying the code of the current implementation, yielding some performance gains as well.

In our security evaluation, we established that the interpreter is secure to memory attacks. We also saw that the interpreter is secure against attacks on the software level, given that the kernel's integrity is guaranteed. Now, we want to hypothesize a bit what would be necessary to run the interpreter even without the certainty of a secure kernel, thus in an untrusted environment like the cloud. We already pointed out why this is desirable in section 4.4.2. There are two main problems with running the interpreter on an untrusted kernel. The first problem is interpreter integrity. The protection of the executed program can always be leveraged

by modifying the interpreter, e.g. by patching the interpreter's code to dump the encryption keys to the kernel log when run. The interpreter by itself can not guarantee that it runs in an unmodified state. The second problem is the key storage. Even if the interpreter's integrity is always guaranteed, i.e. the interpreter always behaves correctly, the kernel can always directly access the debug registers with the cipher key to break the encryption. In all likelihood, these two problems can only be solved by bringing additional trusted hardware components into the picture. Indeed, hardware storage for cipher keys is provided by Trusted Platform Modules (TPM) [46]. TPMs have internal memory, unreachable from the outside, in which keys can be stored. Key storage is currently restricted to asymmetric cryptography, thus the TPM generates a public-private key pair, in which the private half is stored in the TPM. For our purpose, the TPM would need to be extended to also involve symmetric cryptography, and at that not only key storage, but also running the whole encryption in hardware. Then, the interpreter can process the encryption of program data purely through hardware, and cipher key storage in vulnerable CPU registers is dropped.

The interpreter integrity problem could be solved with another of TPM's features: integrity measurement. TPMs can measure hard- and software configuration of a system by creating a hash summary and storing that hash within the TPM. The hash can then by uniquely signed by the TPM to remotely attest that the hard- and software is in a certain state. In the scenario of the interpreter running on a remote server, the execution process could go the following way: First, the user invokes the interpreter remotely, issuing it to execute a program. Then, it has to be verified if the interpreter process runs in an unmodified state. This is done by the TPM hashing the interpreter configuration, uniquely signing that hash, and sending it to the remote user. If the user decides the hash is genuine, he sends his executable and the cipher key back to the server, where the cipher key is stored within the TPM, and execution can begin. Whenever the interpreter needs to de-/encrypt data, the TPM is invoked. Before processing the de-/encryption request, the TPM again has to verify that the interpreter process is still in an untampered state. The interpreter is measured again, and the hash is compared to the hash of the starting configuration. Only if no modifications occurred, the encryption request is processed. Otherwise, the request is denied. After execution, the computation result is passed back to the remote user.

In the described scenario, we silently took some features for granted that are, in its current specification, not supported by TPM. First, we presume that secure communication between the user and the TPM, to exchange measurement hashes and cipher keys, is possible. This requires that the user builds up a secure connection to the TPM. In theory, this could be realized using the RSA key of the TPM to build up a TLS connection. It is not necessary that the connection goes directly to the TPM hardware. Instead, it could be channeled through the interpreter, which handles the connection details. The requirement is that on the remote server, only the TPM can read the connection content. The second assumption we took is that the TPM can measure the interpreter endogenously, i.e. the TPM knows how to correctly measure the interpreter process. In the current TPM design, an application itself provides the measurement hash to the TPM, which opens possibilities for spoof attacks by sending forged hashes to the TPM. We have not looked into how exactly measurements outgoing from the TPM in a tamper-proof way can be implemented. At least, it appears to be a complex problem.

The advantage of running the interpreter in the above way is that the hardware and the operating system of the remote system do not have to be trusted, it is enough if only the interpreter process is verified at runtime. This is in contrast to the traditional approach to achieve secure remote computation, in which the system is authenticated by a so-called chain of trust, which is extended from early boot till the operating system launches. A technology which enables this is Intel TXT [47], which builds up on TPM. However, even after the whole system is running verified, an attacker could still intrude the system by means of a software bug, e.g. a kernel exploit. In the above design, the interpreter can still run in security even after an adversary has achieved ring 0 privileges. All in all, there are many obstacles which must be solved before the interpreter can run securely in a fully untrusted environment, including extending TPM hardware by several more features. Thus, it is unfortunately unsure if the sketched design can be realized in the future. If it can be though, it would by all means be an impactful achievement.

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 could be easily 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. The most prominent is probably the cold boot attack [5, 6], but attacks can also be conducted through peripheral devices by exploiting direct memory access [8, 9]. The accepted protection measure has been to move disk encryption to a lower level, namely to the CPU [13], again. While disk encryption is therewith secure against memory attacks, RAM still contains a lot of unencrypted private data which can be stolen with a memory attack. Executing programs outside memory, using memory only for encrypted data, would protect sensitive user data against memory attacks.

In this thesis, we have shown how encrypted program execution can be done in a memory attack resistant way. 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, where the interpreter processes them. Before any data is transferred from the CPU to memory, it is encrypted first, to ensure confidentiality. As the encryption must be able to resist memory attacks, it can not be done in a conventional way. Thus, TRESOR's cold boot resistant AES implementation was used, which "misuses" the CPU's debug registers for key storage.

We also provide a working proof-of-concept implementation for the x86 architecture, in form of a kernel module for the Linux kernel. Prerequisite for running the interpreter is hardware support in form of the CPU instruction sets AVX for extended registers, and AES-NI for AES hardware acceleration. On the software side, the TRESOR Linux kernel patch is required for key management and key protection. The proof-of-concept implementation of the interpreter supports only a simple bytecode language, but we have shown that the concept can be extended to include more features. Beyond the bytecode language, a higher level language named "SCLL" was developed, to avoid having to program for the interpreter in bytecode. The language includes a compiler which allows generating bytecode programs from SCLL code, as well as encrypting them in a secure fashion.

To analyze the interpreter's resistance against memory attacks, several memory dumps were taken and scanned for patterns of cipher key, 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 and its predecessor AESSE. And the security of TRESOR, as well as AESSE 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 cipher keys. To protect executed programs even against an untrusted kernel, which has broad applications in cloud computing, special trusted hardware components are most likely necessary. Without further measures, the interpreter is vulnerable to a special DMA attack [44], which inserts malicious code into the kernel to obtain the cipher key. This attack can be mitigated by restricting or deactivating DMA.

To conclude, the proof-of-concept implementation is far from deployable in practice – it is merely a prototype. Further development has to be done before the interpreter can truly serve as a general purpose language secure against memory attacks. In the conducted benchmarks, the interpreter performed on average 4 times slower than Python, which is acceptable, considering the heavy overhead through encryption. Thus, from a performance perspective, the interpreter's application in practice is not impeded. The interpreter is securing program execution against memory attacks, as well as software attacks, provided that the kernel's integrity is guaranteed.

Bibliography

[1] Peter Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251327.1251331.

[2] A Guide to Understanding Data Remanence in Automated Information Systems. NCSC-TG-025, National Computer Security Centre, Sep 1991. URL http://fas.org/irp/nsa/rainbow/tg025-2.htm.

[3] Sergei Skorobogatov. Low temperature data remanence in static RAM. Technical Report UCAM-CL-TR-536, University of Cambridge, Computer Laboratory, Jun 2002. URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf.

[4] P. Wyns and Richard L. Anderson. Low-temperature operation of silicon dynamic random-access memories. *Electron Devices, IEEE Transactions on*, 36(8):1423–1428, Aug 1989. ISSN 0018-9383. doi: 10.1109/16.30954.

[5] J. Alex Halderman, Seth D. Schoen, William Clarkson Nadia Heninger, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009. doi: 10.1145/1506409.1506429. URL http://doi.acm.org/10.1145/1506409.1506429.

[6] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In *The Eight International Workshop on Frontiers in Availability, Reliability and Security (FARES 2013)*, pages 390–397. IEEE Computer Society, 2013. URL http://www1.cs.fau.de/filepool/projects/coldboot/fares_coldboot.pdf.

[7] Tilo Müller and Michael Spreitzenbarth. FROST: Forensic Recovery of Scrambled Telephones. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 373–388, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-38979-5. doi: 10.1007/978-3-642-38980-1_23. URL http://dx.doi.org/10.1007/978-3-642-38980-1_23.

[8] Michael Becher, Maximillian Dornseif, and Christian N Klein. FireWire: all your memory are belong to us. In *Proceedings of CanSecWest Applied Security Conference*, Vancouver, British Columbia, Canada, 2005. Laboratory for Dependable Distributed Systems, RWTH Aachen University.

[9] Adam Boileau. Hit by a bus: Physical access attacks with Firewire. In *Proceedings of Ruxcon '06*, page 3, Sydney, Australia, 2006.

[10] Jürgen Pabel. Frozen cache, Jan 2009. URL http://frozenchache.blogspot.com.

[11] Tilo Müller, Andreas Dewald, and Felix C. Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, EUROSEC '10, pages 42–47, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0059-9. doi: 10.1145/1752046.1752053. URL http://doi.acm.org/10.1145/1752046.1752053.

[12] Patrick Simmons. 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 '11, pages 73–82, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076743. URL http://doi.acm.org/10.1145/2076732.2076743.

[13] Tilo Müller, Felix C. Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *Proceedings of the 20th USENIX conference on Security (SEC'11)*, pages 17–17, Berkeley, CA,

USA, 2011. USENIX Association. URL http://www.usenix.org/events/sec11/tech/full_papers/Muller.pdf.

[14] Peter T. Breuer and Jonathan P. Bowen. A fully homomorphic crypto-processor design: correctness of a secret computer. In Jan Jürjens, Benjamin Livshits, and Riccardo Scandariato, editors, *Proceedings of the 5th international conference on Engineering Secure Software and Systems (ESSoS'13)*, pages 123–138. Springer-Verlag, Berlin, Heidelberg, 2013. doi: 10.1007/978-3-642-36563-8__9. URL http://dx.doi.org/10.1007/978-3-642-36563-8__9. http://www.researchgate.net/publication/235342957_A_Fully_Homomorphic_Crypto-Processor_Design_Correctness_of_a_Secret_Computer/file/79e41511199e5b0cc7.pdf.

[15] Chris Lomont. *Introduction to Intel Advanced Vector Extensions*. Intel Corporation, Jun 2011. URL https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf.

[16] National Institute for Standards and Technology. *Announcing the Advanced Encryption Standard (AES)*, federal information processing standards publication 197 edition, Nov 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[17] Johannes Götzfried and Tilo Müller. ARMORED: CPU-Bound Encryption for Android-Driven ARM Devices. In *Proceedings of the 2013 International Conference on Availability, Reliability and Security*, ARES '13, pages 161–168, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-5008-4. doi: 10.1109/ARES.2013.23. URL http://dx.doi.org/10.1109/ARES.2013.23.

[18] M. Brenner, J. Wiebelitz, G. von Voigt, and M. Smith. Secret program execution in the cloud applying homomorphic encryption. In *Digital Ecosystems and Technologies Conference (DEST), 2011 Proceedings of the 5th IEEE International Conference on*, pages 114–119, May 2011. doi: 10.1109/DEST.2011.5936608.

[19] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. URL http://crypto.stanford.edu/craig.

[20] G. Duc and R. Keryell. CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 483–492, Dec 2006. doi: 10.1109/ACSAC.2006.21.

[21] Michael Henson and Stephen Taylor. Beyond Full Disk Encryption: Protection on Security-enhanced Commodity Processors. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 307–321, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-38979-5. doi: 10.1007/978-3-642-38980-1_19. URL http://dx.doi.org/10.1007/978-3-642-38980-1_19.

[22] P.A.H. Peterson. Cryptkeeper: Improving security with encrypted RAM. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 120–126, Nov 2010. doi: 10.1109/THS.2010.5655081.

[23] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 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 '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2118-1. doi: 10.1145/2487726.2488368. URL http://doi.acm.org/10.1145/2487726.2488368.

[24] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, Jun 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL http://doi.acm.org/10.1145/857076.857077.

[25] M. Anton Ertl and David Gregg. The Structure and Performance of *Efficient* Interpreters. *The Journal of Instruction-Level Parallelism*, 5, Nov 2003. URL http://www.jilp.org/vol5/v5paper12.pdf. http://www.jilp.org/vol5/.

[26] *Labels as Values*. GNU. URL https://gcc.gnu.org/onlinedocs/gcc-3.1/gcc/Labels-as-Values.html.

[27] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs*. Technical University of Denmark, Aug 2014. URL http://www.agner.org/optimize/microarchitecture.pdf.

[28] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. In *Proceedings of the 17th International Conference on The Theory and Application of Cryptology and Information Security*, ASIACRYPT'11, pages 344–371, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25384-3. doi: 10.1007/978-3-642-25385-0_19. URL http://dx.doi.org/10.1007/978-3-642-25385-0_19.

[29] Shay Gueron. *Intel Advanced Encryption Standard (AES) New Instructions Set*. Intel Architecture Group, Israel Development Center, Intel Corporation, revision 3.01 edition, Sep 2012. URL https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set.

[30] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key Recovery Attacks of Practical Complexity on AES-256 Variants with Up to 10 Rounds. In *Proceedings of the 29th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'10, pages 299–319, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13189-1, 978-3-642-13189-9. doi: 10.1007/978-3-642-13190-5_15. URL http://dx.doi.org/10.1007/978-3-642-13190-5_15.

[31] dm-crypt: Linux kernel device-mapper crypto target. URL https://code.google.com/p/cryptsetup/wiki/DMCrypt.

[32] TrueCrypt Foundation. TrueCrypt. URL http://truecrypt.sourceforge.net/.

[33] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual Machine Showdown: Stack Versus Registers. *ACM Trans. Archit. Code Optim.*, 4(4):2:1–2:36, Jan 2008. ISSN 1544-3566. doi: 10.1145/1328195.1328197. URL http://doi.acm.org/10.1145/1328195.1328197.

[34] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A-Z*, Sep 2014. URL http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[35] Agner Fog. *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*. Technical University of Denmark, Feb 2014. URL http://www.agner.org/optimize/optimizing_assembly.pdf. Section 13.6: "Using AVX instruction set and YMM registers".

[36] National Institute for Standards and Technology. *Recommendation for Block Cipher Modes of Operation*, NIST Special Publication 800-38A edition, Dec 2001. URL http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

[37] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security Analysis of Pseudo-random Number Generators with Input: /Dev/Random is Not Robust. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 647–658, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516653. URL http://doi.acm.org/10.1145/2508859.2516653.

[38] Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau. The Linux Pseudorandom Number Generator Revisited. *Cryptology ePrint Archive*, (Report 2012/251), 2012. URL https://eprint.iacr.org/2012/251.pdf.

[39] Stephan Müller, Gerald Krummeck, and Mario Romsy. Dokumentation und Analyse des Linux-Pseudozufallszahlengenerators. Technical Report Version 3.8, Bundesamt für Sicherheit in der Informationstechnik, Dec 2013. URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Studien/LinuxRNG/LinuxRNG_Studie.pdf.

[40] Suresh Siddha. x86: add linux kernel support for YMM state, Apr 2009. URL http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a30469e7921a6dd2067e9e836d7787cfa0105627.

[41] Floating-point support for 64-bit drivers. URL http://msdn.microsoft.com/en-us/library/ff545910.aspx.

[42] Fabrice Bellard. QEMU: Open Source Processor Emulator. URL http://www.qemu.org.

[43] KVM: Kernel Based Virtual Machine. URL http://www.linux-kvm.org/.

[44] Erik-Oliver Blass and William Robertson. TRESOR-HUNT: Attacking CPU-bound Encryption. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 71–78, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2420961. URL http://doi.acm.org/10.1145/2420950.2420961.

[45] James Reinders. *AVX-512 instructions*. Intel Corporation, Jul 2013. URL https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[46] ISO/IEC 11889-1. Information technology – Trusted Platform Module – Part 1: Overview, 2009. URL http://www.iso.org/iso/catalogue_detail.htm?csnumber=50970.

[47] James Greene. Intel Trusted Execution Technology: Hardware-based Technology for Enhancing Server Platform Security. Technical report, Intel Corporation, 2012. URL http://www.intel.de/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf.

A

APPENDIX

A.1. SCLL Grammar

Listing A.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] ;
argdef        =  type | identifier ;
arglist       =  argdef, [',', arglist] | 'void' ;
func          =  type, identifier, '(', arglist, ')', ('{', sequence, '}' | ';') ;
program       =  [func, program] ;
```

A.2. Source Codes

In this section, the SCLL source code of programs that were used in this thesis is listed. In particular, the programs A.2, A.3, and A.4 were used in benchmarking, whereas program A.5 was applied in scanning memory dumps for suspicious byte patterns.

Listing A.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(void) {
        int n = 35;
        print fib(n);
}
```

Listing A.3: Program calculating the primes to `max_prime`.

```
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(void) {
        int primes = 1000000;
        for(int i = 2; i <= primes; i = i + 1) {
                print_prime(i);
        }
}
```

Listing A.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(void) {
        int max_row = 23;

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

```
                }
        }
}
```

Listing A.5: Program adding random integers, used in memory scan.

```
int random(void) {
        return
        0xfa224713+0x95716f55+0x721fa3b8+0x19a099c1+
        0xffc643d2+0xc51fd9cf+0x18624988+0xaa6095e3+
        0xb944d61c+0x0c11da31+0x9dff0b70+0x6d82c2c6+
        0x91a2e74e+0x74f88940+0xd4b534c1+0x5c97b451+
        0xa89cf9e0+0x19c5c71d+0xeaafb801+0x62f6eeab+
        0xa24019d3+0xaf981a6c+0x30a1a9b7+0x5bc3dfe6+
        0x9071048b+0x0a1bd736+0x4c9fa643+0x12e428a4+
        0xcce755e8+0xee43ca56+0x0ebfdf71+0xbc4580a9;
}

void main(void) {
        print random();
}
```

A.3. Memory Scan Patterns and Results

In this section, the different patterns used in the memory scans are listed. Besides each pattern, the scan result in form of the longest matching byte sequence is given. Where the longest found sequence has the same length for little and big endian search pattern, only the little endian result is given for brevity. If the length differs, both are specified.

Listing A.6: AES round key search.

```
rk0:  084e63eef35200201a7b7489d2514ae4        longest match: 3 bytes, '200052'
                            big endian        longest match: 4 bytes, '00201a7b'
rk1:  74dd2c0f79808aaf6c797190272a4e6e        longest match: 3 bytes, '71796c'
rk2:  7688553f83569ade23c6413220ef7428        longest match: 3 bytes, '3241c6'
rk3:  460e995e4e40fab0bd12fa90a7698e19        longest match: 3 bytes, '69a790'
rk4:  ac5c14a3dad4419c5982db427a449a70        longest match: 3 bytes, '599c41'
rk5:  18a45d9f5eaac4c110ea3e71adf8c4e1        longest match: 3 bytes, 'c4f8ad'
rk6:  f80758d6545b4c758e8f0de9d70dd6ab        longest match: 3 bytes, 'd60dd7'
                            big endian        longest match: 4 bytes, '0de9d70d'
rk7:  0d5da6a015f9fb3f4b533ffe5bb9018f        longest match: 3 bytes, 'fe3f53'
rk8:  2292eb01da95b3d78eceffa20041f24b        longest match: 3 bytes, 'b395da'
rk9:  e4492449c6dbcf481c4e7c9f9280833d        longest match: 3 bytes, '3d8380'
rk10: 7d67a27809ba8e77703a04d81c437548        longest match: 3 bytes, '778eba'
rk11: 53d92e1fb7900a56714bc51e6d05b981        longest match: 3 bytes, 'b71f2e'
rk12: d849688ea52ecaf6ac944481dcae4059        longest match: 3 bytes, 'aedc81'
rk13: 993c1dcfcae533d07d7539860c3efc98        longest match: 3 bytes, 'd033e5'
rk14: 87b34d445ffa25cafad4ef3c5640abbd        longest match: 3 bytes, '5f444d'
```

Note that the AES round key search patterns are given in big endian here.

Listing A.7: Fib program, bytecode search.

```
120000003100000001000000014000000    longest match: 11 bytes, '00000001000000014000000'
010000000400000002000000002000000    longest match: 15 bytes, '000000040000000200000002000000'
                      big endian     longest match: 16 bytes, '0000000200000002000000400000001'
010000000d0000001000000002000000     longest match: 11 bytes, '000000100000002000000'
010000001500000001000000013000000    longest match: 12 bytes, '010000001500000001000000'
040000000200000002000000002000000    longest match: 16 bytes, '040000000200000002000000002000000'
0d0000001b0000000200000001000000     longest match: 11 bytes, '0000001b00000002000000'
                      big endian     longest match: 12 bytes, '0000000100000020000001b'
150000000100000013000000004000000    longest match: 11 bytes, '00000001000000013000000'
020000000200000001000000007000000    longest match: 12 bytes, '020000000200000001000000'
050000000000000012000000003000000    longest match: 13 bytes, '0500000000012000000003000000'
                      big endian     longest match: 12 bytes, '000000120000000000000005'
040000000200000002000000002000000    longest match: 16 bytes, '040000000200000002000000002000000'
070000000500000000000000012000000    longest match: 12 bytes, '070000000500000000000000'
                      big endian     longest match: 13 bytes, '0000001200000000000000000500'
030000000600000015000000001000000    longest match: 11 bytes, '000000060000001500000'
130000001400000001000000002000000    longest match: 11 bytes, '00000001000000002000000'
                      big endian     longest match: 12 bytes, '000000020000000100000014'
230000000500000000000000012000000    longest match: 12 bytes, '000000050000000000000012'
                      big endian     longest match: 13 bytes, '0000001200000000000000000500'
030000000300000015000000001000000    longest match: 11 bytes, '000000150000000001000000'
```

Listing A.8: Random program, bytecode search.

```
134722fa02000000556f719502000000        longest match: 6 bytes, '22fa02000000'
                      big endian        longest match: 5 bytes, '0000000295'
b8a31f7202000000c199a01902000000        longest match: 6 bytes, 'a01902000000'
                      big endian        longest match: 5 bytes, 'c100000002'
d243c6ff02000000cfd91fc502000000        longest match: 5 bytes, 'c502000000'
88496218020000000e39560aa02000000       longest match: 6 bytes, '60aa02000000'
                      big endian        longest match: 5 bytes, '0000000218'
1cd644b90200000031da110c02000000        longest match: 6 bytes, 'b90200000031'
                      big endian        longest match: 5 bytes, '00000002b9'
700bff9d02000000c6c2826d02000000        longest match: 6 bytes, 'ff9d02000000'
                      big endian        longest match: 5 bytes, 'c600000002'
4ee7a291020000004089f87402000000        longest match: 5 bytes, '0000004089'
c134b5d40200000051b4975c02000000        longest match: 5 bytes, 'd402000000'
                      big endian        longest match: 6 bytes, '00000002d4b5'
e0f99ca8020000001dc7c51902000000        longest match: 5 bytes, 'a802000000'
01b8afea02000000abeef66202000000        longest match: 6 bytes, 'f66202000000'
                      big endian        longest match: 5 bytes, '00000002ea'
d31940a2020000006c1a98af02000000        longest match: 5 bytes, '020000006c'
b7a9a13002000000e6dfc35b02000000        longest match: 5 bytes, '02000000e6'
8b0471900200000036d71b0a02000000        longest match: 5 bytes, '0a02000000'
43a69f4c02000000a428e41202000000        longest match: 5 bytes, '1202000000'
                      big endian        longest match: 6 bytes, '0000000212e4'
e855e7cc0200000056ca43ee02000000        longest match: 6 bytes, 'e7cc02000000'
                      big endian        longest match: 5 bytes, '5600000002'
71dfbf0e02000000a98045bc06000000        longest match: 6 bytes, '45bc06000000'
                      big endian        longest match: 5 bytes, 'a900000002'
```

Listing A.9: Random program, stack data search.

```
fa224713957l6f55721fa3b819a099c1        longest match: 3 bytes, '19b8a3'
ffc643d2c51fd9cf18624988aa6095e3        longest match: 3 bytes, 'd91fc5'
     after 1000 runs, big endian        longest match: 4 bytes, '6d82c2c6'
b944d61c0c11da319dff0b706d82c2c6        longest match: 3 bytes, '826d70'
91a2e74e74f88940d4b534c15c97b451        longest match: 3 bytes, '51b497'
a89cf9e019c5c71deaafb80162f6eeab        longest match: 3 bytes, '6201b8'
a24019d3af981a6c30a1a9b75bc3dfe6        longest match: 3 bytes, '1940a2'
9071048b0a1bd7364c9fa64312e428a4        longest match: 3 bytes, 'd71b0a'
cce755e8ee43ca560ebfdf71bc4580a9        longest match: 3 bytes, 'eee855'
```

Note that the Random program stack data search patterns are given in big endian here.