FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Towards A Write ⊕ Execute Architecture For JIT Interpreters

Lobotomy

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Martin Jauernig
Matrikelnummer 0225818

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao. Univ. Prof. Dr. Wolfgang Kastner
Mitwirkung: Univ. Ass. Dr. Christian Platzer
Dr. Paolo Milani Comparetti

Wien, 04.07.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Towards A Write ⊕ Execute Architecture For JIT Interpreters

Lobotomy

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Martin Jauernig
Registration Number 0225818

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao. Univ. Prof. Dr. Wolfgang Kastner
Assistance: Univ. Ass. Dr. Christian Platzer
Dr. Paolo Milani Comparetti

Vienna, 04.07.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Martin Jauernig
Heiderichstrasse 3/4/11, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Abstract

JIT spraying is one of the main reasons why current web browsers still provide a large attack surface for remote exploits. It is capable of circumventing even the most sophisticated defense strategies against code injection, including address space layout randomization (ASLR), data execution prevention (DEP) or stack canaries. In this thesis, we present Lobotomy, an architecture for building injection-safe JIT engines by splitting compiler and executor into two different processes. We also present a proof-of-concept implementation of our approach by modifying the well-known JIT-Engine Tracemonkey as it is shipped with the Firefox Browser. Additionally, we provide a thorough evaluation of performance and code coverage of our version compared to the unmodified baseline.

Kurzfassung

Verschiedene Fortschritte im Bereich Software-Sicherheit, vor allem der großflächige Einsatz von Technologien wie ASLR, nicht ausführbarer Speicher, stack-cookies und ähnlichen Mechanismen haben die Durchführung von Angriffen auf Software-Systeme über Puffer-Überläufe deutlich erschwert. Andererseits werden Web-Browser auf Grund der zunehmenden Verlagerung von Applikationen auf externe Server immer bedeutender. Um potentiell interaktive Inhalte, die oft mit Javascript und anderen Scriptsprachen implementiert werden, in akzeptabler Zeit anzeigen zu können, müssen Web-Browser aber einige der oben genannten Sicherheitsvorkehrungen deaktivieren. Im besonderen werden einige Bereiche des Heaps als ausführbar markiert, um den dort generierten Maschinencode ausführen zu können. Dies erlaubt sogenannte Just-In-Time (JIT) Heap-Spraying-Attacken. Im Zuge dieser Diplomarbeit wird ein Mechanismus vorgestellt, der solcherart Angriffe er- schwert. Dabei wird ein JIT Interpreter (im konkreten TraceMonkey von Firefox 5.0) in zwei unterschiedliche Prozesse aufgespalten, die zu keinem Zeitpunkt sowohl Schreib- als auch Aus- führungsberechtigung für eine Speicherseite haben. Dabei wird sichergestellt das das auszuführende Skript in einem anderen Kontext läuft als der Rest des Interpreters. Damit wird die Angriffsfläche die zur Durchführung einer Attacke auf den Interpreter verwendet werden kann reduziert. Desweiteren wird die modifizierte Version des Interpreters auf ihr Laufzeitverhalten sowie ihren Speicherverbrauch evaluiert. Zur Überprüfung der Geschwindigkeit wird ein von Mozilla zur Verfügung gestelltes Benchmark-Werkzeug verwendet. Der Speicherbedarf wird mit Hilfe der standard- mäßig unter Linux installierten Werkzeuge ermittelt. Anschließend werden die erhobenen Daten mit einer unmodifizierten Version von TraceMonkey verglichen. Ausserdem wird die Resistenz der neuen Architektur gegenüber Angriffen auf den JIT-compiler getestet.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Advances in software security made in the last years, in particular the widespread implementation of Address Space Layout Randomization (ASLR) [34], non-executable memory (NX), stack cookies and similar technologies have made the exploitation of vulnerabilities in popular and well-audited software increasingly difficult. In response, attackers have developed several advanced techniques in order to circumvent as many countermeasures as possible. These techniques include Return To Lib(C) attacks, Return Oriented Programming (ROP) [8] or Just-In-Time Heap Spraying (JIT-spraying) [5].

Compared to the other recent attacks, JIT-spraying has the advantage that an attacker can 'spray' the target with code of the attacker's design, generated by the JIT-engine, instead of being forced to use already existing code (which requires precise jumps that may be difficult to carry out when addresses are randomized). The 2012 Pwn2Own contest [13] has impressively shown the potential of JIT-spraying, when Firefox was compromised by a vulnerability that enabled such an attack.

Using JIT-spraying, an attacker can readily bypass any non-executable memory protections that may be used, since the JIT-engine has to allocate its code-pages with both write and execute permissions per design. This behavior is necessary to ensure simplicity and especially efficiency, but is obviously somewhat risky. Furthermore, it is incompatible with systems such as SELinux, which enforce a rigid W $\oplus$ X (Write $\oplus$ eXecute) regime. ASLR and similar mechanisms are weakened due to the fact that the attacker can cause the engine to compile large amounts of code which may contain a sled of semantic NOPs. Additionally, randomization is negated entirely should an attacker find an address-disclosure vulnerability. The ultimate result is a reliable attack that uses the compilation facilities of a target against it.

The root of JIT-spraying is the presence of memory that is both writable and executable within the address-space of a process. To counter this attack, one has to find a way to separate the write-permissions from the execute-permissions, either temporally or spatially. One approach is to allocate two regions of memory backed by the same temporary file, separated by a certain offset within the same process. This is a stable and fast solution. However, should an attacker

discover this (not explicitly randomized) offset, any security gained by implementing such a mechanism would be lost again [21].

Another possible approach is to change the mapping on demand. A memory region would be mapped writable and only marked as executable when the compiled code needs to be run. However, if an attacker could stall, delay or lengthen the execution of compiled code while the associated pages are marked executable, a successful exploitation is still feasible [9]. Additionally, this approach may leave memory-regions unprotected that contain compiled code but are not currently being executed. Furthermore, the frequent re-mapping of many disjoint memory segments is very time-consuming as it causes a constant flood of syscalls and the involved changes on the hardware side (such as frequent updates of the Translation Lookaside Buffer).

In the course of this thesis, we will describe a new, general approach, tested using Firefox and its JIT-engine Tracemonkey. Instead of separating writable and executable memory by time or by an offset within the memory-space of a process, the two regions are placed in two different processes. As a result, an attacker exploiting the process with write-permissions has no access to executable codepages. With this method, a vulnerability in the process where the codepages are mapped executable might still lead to a compromise. However, we believe that by confining execution of JIT-code to a dedicated process with a limited number of lines of code involved, the problem will become easier to control. Additionally, a temporal separation as proposed in [9] may be enforced on the executing process, thereby reducing the surface for successful attacks even further. In this case, a tradeoff between security and speed has to be found.

To summarize, our contributions are threefold:

- We introduce a novel architecture for JIT-engines to improve their resilience against attacks that employ JIT-spraying.
- We present a proof of concept implementation of this archicture using TraceMonkey as it is shipped with Firefox 5.
- We discuss our results and evaluate our approach and the impact this architecture has on the JIT-engine.

1.1 Methodology

The primary goal of the research-project this thesis is based on was to prove or disprove the hypothesis that introducing a process-boundary between the main body of a virtual machine and the component that executes dynamically generated code increases its security against attacks in which dynamically generated code is accessed through a rogue jump or function-call. The counter-hypothesis in this case states that no additional security is gained over an architecture which differs from the unmodified engine only in the initial access-rights to the memory-pages holding the dynamically compiled code (an approach similar to Jitdefender [9]).

To test our hypothesis, we make use of traditional Software- and Reverse-Engineering concepts. We designed an architecture which we then implemented. The implementation was tested and the results were evaluated. The results of the evaluation were again used to refine the design

of the architecture, thus restarting the cycle. Last outcome to date, as well as excerpts from critical intermediate stages, is presented in this thesis.

Initially, a first proof-of-concept in which only the JIT-compiler (`nanojit`) itself was modified was developed. Once that prototype was operational, the same concept was applied to the entire virtual machine. The linux debugger `GDB` [33] was used to develop an understanding of the internals of this large and complex system. Each implementation was tested both manually and by running a benchmark- and regression-testing suite packaged with the javascript-engine. The presently available version was also tested with the binary instrumentation tool `PIN` to determine the code-coverage of both parts of the engine. The outcome of these tests is shown in this thesis.

1.2 Previous Work

The initial prototype mentioned above was developed as part of an internship in February 2011. The latest version of the architecture was submitted to the Usenix- and CCS-conferences where valuable Input from reviewers was received. It is now being submitted to NDSS. Therefore, parts of this thesis were written for research-papers submitted to these conferences.

CHAPTER 2

Background

2.1 Memory Corruption

A critical problem with the architecture of modern computers is their inability to distinguish between code to be executed and data to be processed. The processor of a computer has an instruction pointer that increases with every executed instruction and that can be modified by the `jump` and `call` instruction-families. The memory-address pointed to by the instruction pointer is implicitly assumed to contain valid instructions in a format understood by the processor. It falls to the compiler-software, or in case of a program written using a symbolic assembler, the programmer, to ensure that this assumption holds. This is not always trivial, especially since a copy of the current instruction pointer is often written to memory when a pointer to a different section of executable code is required. One such case is the call of a function. To call functions, the a stack is built within the memory occupied by the process. This stack consists of frames, each of which corresponds to the invocation of a function. A frame contains the function's local variables. It may also contain any parameters that cannot be stored in a register. Additionally, it will contain several pointers that reference the last frame which enable or ease the return from the function and the destruction of the corresponding stack-frame. One such pointer is called `EIP`, the stored instruction pointer (`IP`). This field contains the next address after the invocation of the function the frame belongs to. When the function finally returns, this address is read out and the value stored there replaces the current instruction pointer. There are also several other pointers that reference executable code, such as exception handlers or the entries in the global offset table used to dynamically load functions.

The instruction-pointer obviously plays a central role in the execution of a program and while a careless application may simply edit it, more carefully written software will typically avoid such behaviour or only do so in a clearly defined and safe manner. However, since the `EIP` is used to restore the instruction pointer after a return from a function, similar care is necessary to ensure the integrity of each stored instruction pointer. Unfortunately, many programs in the past fell short at this point. Nothing except common-sense prevents a process from writing to the saved instruction-pointer of a function and editing it. Since compilers typically try to

produce code that is as sane and safe as possible, they will never generate code that attempts to do this. Yet, writes to addresses that are near to the critical address that contains the `EIP` are perfectly valid, since local variables can be stored there. It is the task of the compiler to ensure that enough space is reserved on the stack to hold each local variable in a stack-frame without overflowing into other areas of the stack (such as the `EIP`). Given that variables in strongly typed languages such as C/C++, which are still frequently used when execution-speed is of importance, have a fixed size, preventing them from overflowing is usually easy for a compiler. However, a variable may be a buffer, often also called array, that is, a continuous region of memory which may be written to in an arbitrary fashion, usually by specifying the base-address and an offset (the array-index). The question of whether or not an access to such an array of memory-cells is valid or not is made more complex by the fact that it is possible to store pointers to addresses other than the original base-address and to use these pointers as base-address instead. Higher-level programming-languages solve this problem by creating an object for each array with a length-attribute assigned to it. A basic array of memory-cells does not have this. The compiler can therefore usually not tell whether an access is valid or not. This allows for data that would belong into the array to overflow and be written into cells reserved for potentially critical management-information. Operations that copy bulk-data are notorious for triggering such bugs. For example, the C-function `strcpy()` writes data read from the address supplied as the second argument to the address passed as the first argument. It copies the data byte by byte, until it reads a byte with the value null. It does so without checking the bounds of the variable stored at the destination-address. Since the stack grows downwards in memory and the various stored pointers that are attractive targets for overwriting if an attacker wants to control the flow of the program are located above the local variables, `strcpy()` can overwrite any critical information stored on the stack above the overflowing variable. The amount of data that can be written only depends on the amount of non-null bytes located at and after the source-address. If the buffer at the source-address can hold more data than the buffer at the destination and the contents of the source-buffer are controlled by an attacker, the attacker can overwrite the `EIP` of the current function with an arbitrary value. Once the function returns, the `EIP` will replace the current instruction pointer and the processor will execute any instruction located at the address supplied by the attacker. Both, the pointer and the 'code' to be executed, are in this scenario introduced as user-data. Since the processor knows no difference between user-data, pointers and code, it is executed anyway. The attacker can now try to guess which address to write to `EIP`, or, if possible, the attacker may also try to exploit proximity of the injected malicious code to certain known addresses, such as the top of stack or the pointer to the current frame. The ultimate aim of the attacker is for the overwritten pointer to reference an address that is controlled by the attacker. Typically, that is the beginning of the buffer that triggered the overflow. This attack was first described and made popular by Aleph One [23] and has since been known as stack-based buffer-overflow.

Listing 2.1 shows a function that is vulnerable to a simple buffer-overflow. The first line of the function defines a buffer that is 16 bytes long. On a 32-bit system four fields, each four bytes long, will be reserved on the stack to hold the buffer. On the second line, the contents of the variable `overflow` are copied to the previously allocated buffer. However, no bounds-checks are performed here by default. If the buffer pointed to by `overflow` is longer than 16 bytes,

the data contained in it will spill over into another field on the stack.

Figure 2.1 shows the stack before the second line is executed. The base-address of the buffer, currently residing at the top of the stack, is given as 0xBF000000. The beginning of the stack-frame, located 24 bytes higher in memory is at 0xBF000018 and contains the stored return address. Immediately below that is a stored frame pointer. The frame pointer may be used to conveniantly locate variables even when the stack pointer, which refers to the top of the stack, changes during the execution of the function (i.e when additional variables are declared after the start of the function or undeclared before the end). The field of the same name contains the value of the frame pointer when the function was called. This pointer may be a target of a buffer-overflow in certain cases, particularly when it is for some reason used as a base-address for a function-call or a jump.

Figure 2.2 shows the stack-frame of the function after the `strcpy()`-call. Instead of respecting the bounds of the 16 bytes long buffer, the attacker has passed the vulnerable function a pointer to a buffer that contains 24 bytes of data. The tail end of that buffer has overwritten both the saved frame pointer and the saved instruction pointer. In this case, the overflowing buffer contained 20 bytes with a value of 0x90, which is interpreted by an IA32-processor as 'No Operation', or NOP. An attacker may fill large amounts of memory with these instructions to create a sled that leads to the actual payload of the exploit. This is desirable for the attacker because it alleviates the need to hit the exact beginning of the payload for a successful attack. The last four bytes of the buffer contain the beginning of the attacker-supplied code. The example given here would, of course, have no effect since the return address was overwritten with a pointer to what is now a block of code that does nothing. Given the total absence of bounds-checking in this function, an attacker would likely overwrite the return address with a pointer to a region above the current stackframe. There, yet more NOPs would be placed, followed by the actual payload, hundreds or thousands of bytes further up in memory.

Listing 2.1: Simple Buffer Overflow

```
1 void vulnerable(char *overflow){
2         char buffer[16];
3         strcpy(buffer, overflow);
4 }
```

Other, more sophisticated attacks have since been developed. For example, buffers located on the heap can be caused to overflow as well. The target, in this case, is not the `EIP` or a similar code-pointer but pointers to another heap-bucket (for heap-memory as allocated by the `malloc` family of heap-management functions. These pointers control the destination of a write performed when modifying the heap (allocating new buckets or freeing them). Among the data written into the destination-address is the length of a bucket. If the attacker can set the destination-address in such a way that the attacker-supplied 'length' (actually a pointer) is written to a code-pointer such as `EIP`, the attacker can take control of the process. Both of the above-mentioned situations are exploitable due to insufficient bounds-checking. That is, the programmer failed to correctly specify and limit the amount of data that is to be written to a buffer. LibC provides utility-functions such as `strncpy`, which take an additional length-argument that specifies the amount of data to be copied at most (not including the terminating

null-byte). However, even these functions can be missused. For example, programmers have been known to mis-caluclate the length. One common mistake is to use signed integer variables for such calculations. If a small negative integer is cast as an unsigned integer, the result is a very large number, negating the security gained by adding a length-argument to the `str*`-functions.

Another vector of attack are use-after-free vulnerabilities. Unlike buffer-overflows, which depend, in part, on the absence of type-information about the length of an object, use-after-free attacks exploit type-information which is no longer valid. The basic principle in this case, is that an attacker causes an object on the heap to be freed, even though the application still has a reference to the object in memory. In many strongly-typed programming-languages such as C++, each class has a list of functions that can be applied to an object (called dispatch-table). These functions are accessed through pointers located at a specific index. If a subclass is created that defines additional methods, the methods are added to the end of the table. Each object of a class contains a pointer to this dispatch-table as its first or last element. This pointer is referred to as the virtual table pointer. If an attacker can change the contents of the object by accessing it after it has been freed, the virtual table pointer can become corrupted and can be replaced by a pointer to an area of memory controlled by the attacker. The data stored therein will be used as function-pointer. If the function-pointer now points to executable memory that has been populated by the attacker, or to a section of benign code that can be supplied with malicious parameters, the system will be compromised. This is the vector commonly used to carry out JIT-spraying attacks. Recently, tools such as `AddressSanitizer` [29] have been used to automatically uncover many such bugs, including bugs in high-profile targets such as browsers.

2.2 Return into LibC and other code-reuse attacks

The malicious code an attacker wants the target to execute does not need to be supplied in a buffer. Instead, already existing, legitimate code can serve as the jump-target of an exploit. This code may be found in shared libraries such as the LibC or in other locations. This attack-technique was first presented in 1997 by a contributor to the Bugtraq mailing-list known as 'Solar Designer' [12]. The author proposed to fix the vulnerability by forcing addresses within shared libraries to always contain a null-byte so that they may not be passed within an overly long string. The problem with this fix is that the target does not have to be located in a shared library. More recent works [30] have generalized this technique into a class of attacks called 'Return Oriented Programming' or 'ROP'. ROP has become increasingly popular in the last years and is now commonly used in attacks found in the wild. As in the generic memory-corruption attacks discussed above, the fundamental idea of this scenario is that executable code such as a program or a shared library, at its most basic level, is a section of memory that contains bytes which cause the processor to act in a certain way. These bytes can be grouped in functions and functions can again be grouped in libraries. However, these groups are not 'natural' constructs but were introduced to increase usability and reusability. A jump into a random area of (executable) memory causes the processor to execute the data found within as if it were the start of a control-block. In reality, however, the bytes that are interpretered as code could just as well be data or they could have been written with an entirely different meaning in mind. Unfortunately, the meaning of a code-byte can be dependent on the context of the byte, since many instructions

consist of multiple bytes. This principle will also be of critical importance to JIT-spraying attacks, discussed later on. In ROP, an attacker searches for 'gadgets'. Gadgets are small sets of instructions that can be found in all places that contain executable memory. To be eligable as a gadget for ROP, a set of instructions need to end with a 'ret'-instruction. This return-instruction consumes a value from the stack and jumps to this value. That way, an attacker can chain together several gadgets. The instructions before the 'ret' of each gadget can be used to perform arbitrary operations. Shacham has shown that this forms a turing-complete sub-language of assembly [30]. Regardless of the expressive power of the 'language' used for it, ROP remains, to this day, a serious threat to the security of computers, more so since gadgets can be extracted automatically [20]. Attackers could even attempt to reuse larger segements of code by automatically extracting parts of it [6].

These attacks and others, as summarized by Payer [25] are still cause for concern among software-developers and security specialists alike. On the other hand, users and developers can choose from a wide array of countermeasures.

2.3 Non-Executable Stack

One approach to combatting memory-corruption attacks is to restrict execute-rights to certain areas of memory. The memory-space allocated for a process is divided into several sections. One section is the code-section. This area of memory contains the executable code as written by the compiler or assembler. It obviously needs to be executable, otherwise the process would be unable to run at all. There are also areas reserved for the storage of data, such as the above mentioned stack and heap. Since these areas are meant for data alone, there is no good reason for the contents of these sections to be executable. While some applications may wish to allow certain parts of the heap to be executed (such as a JIT-compiler writing code on demand), a process should not have need to execute data stored at these locations by default. In particular, there is no sane reason why executable code should reside on the process' stack, which changes whenever a function is called. Nontheless, many popular operating systems allowed execution of data on the stack by default. This made stack-based buffer-overflows very easy to exploit. To harden applications against this class of attacks, both processor-vendors and kernel-developers implemented ways to prevent these attacks. Most modern processors ship with a non-executable bit which, if set, prevents the CPU from executing data in areas of memory marked as non-executable. While applications can set the memory-protection flags directly, Operating System support is needed for these flags to have effect. On Windows, applications need to enable Data Execution Prevention (unless globally enabled by the Administrator). For Linux, there is the PaX kernel patch which makes use of the processor's NX-bit (if available) or emulates the feature if not available, to prevent the execution of data-only areas such as the stack, or any other area flagged as non-executable. MacOS X uses the NX-bit since the Intel-compatible architecture was adopted.

2.4 ASLR

Adress Space Layout Randomization is a technique that is used to increase the security of a system versus attacks that seek to exploit memory-corruption errors. The basic aim of ASLR is to make it more difficult to jump to previously injected malicious code in order to execute it. Without ASLR in place, a variable in memory (such as an array of characters) typically starts at a fixed point in the memory-space because the process itself is always loaded into the same place. This makes it easy for attackers to jump to previously injected malicous code. They can simply start a debugger and find the address their executable 'payload' was written to. They can then code this address into their exploits, confident that it will most likely not change, even when run on a different system. If, however, the start-address of the program is randomized, the attackers can no longer be certain that the jump will meet its mark and cause execution to continue within a buffer filled with malicious code. In fact, the target of the jump will most likely be an invalid address and the programm will simply crash. The addresses into which dynamically linked libraries are loaded can be randomized as well. Without this feature, a program remains vulnerable to attacks such as Solar Designer's Return into LibC [12] attack and its successor, Return Oriented Programming. If ASLR is used correctly, it can greatly increase the security of a system. However, several ways have been found to bypass ASLR and to negate the security gained through it. The simplest countermeasure against ASLR in the attackers arsenal is Heap Spraying. The goal of the attacker in this scenario is to place as much executable payload on the heap as possible. While address of a variable on the heap is still randomized, beyond a certain point the exact address no longer matters since a large section of the heap contains nothing but malicious payload. That way, the attacker gets away with choosing a general location to jump to, instead of having to specify a precise address. This is particularly easy on 32-bit systems which limit the entropy available to the Operating System when randomizing an address. On 64-bit systems, there are far more start-addresses to choose from, making Heap-Spraying much more difficult [31]. Nontheless, Heap-Spraying attacks have gained great popularity among attackers. Worse, Roglia et al [27] showed that in some cases (namely in the absence of a randomized start-address for the process-image) is is possible to extract the address of a (randomized) library-function from the global offset table (GOT). Also, some Operating Systems may, for whatever reasons, not randomize certain areas of memory. For example, it was shown by Yang Yu in [36] that Windows does not randomize the `SharedUserData`-region which starts at 0x7ffe0000 and contains a function that is used to call system-calls. It was recently shown in [19] that ASLR can be bypassed by using a script-interpreter to fill up the heap entirely. Thus, the entropy available for loading additional libraries is reduced to nothing. If just enough memory is now freed to contain the targetted library (i.e. a dll-file corresponding to an ActiveX component), the attacker can predict the address into which the targetted code will be loaded.

Despite being not the end-all solution to memory-security, ASLR, especially in combination with non-executable stack memory can be considered a critical line of defense when securing an information system.

2.5 Canaries and other pointer protection

Another generic countermeasure against attacks based on the overwriting of memory is to insert a specific flag before a critical pointer and to check status of that flag before the pointer is used [10, 15]. When an out-of-bounds write (such as during a buffer-overflow) occurs, it is not only the critical pointer that was targetted by the attacker that is being overwritten. Instead, any variables, pointers or other data between the end of the buffer and the target or targets are also overwritten. Typically, an attacker will insert padding such as NOP-instructions, the instruction 0xC0 (in case the target of a jump needs to be a valid address) or even 'A'-characters to fill the buffer and to reach the target. If, however, a special variable is inserted by the compiler before every pointer that is considered a possible target for a buffer-overflow, and the value of this variable is compared with a reference-value before each usage of the pointer, the system can detect that an out-of-bounds write has occured, since the actual value of the 'canary'-variable differs from the expected value. This flag is referred to as 'canary' in homage to the canary-birds used by miners of previous times to detect suffocating gas.

By default, a compiler using this technique may only protect some very common targets such as `EIP`, function-pointers or pointers to exception-handlers. However, there have been attacks that have targetted other pointers (such as file-pointers, the maintenance-pointers used by the `malloc()` function-family or even other buffers). While protecting all pointers would further increase the security of the system, the status-checks required to do so could potentially cause a noticable performance-overhead.

Naturally, inserting a canary to protect a pointer from being overwritten is only effective if the attacker cannot reproduce the canary. If the canary-flag can be feigned by the attacker, it would be possible to overwrite the flag with itself just before overwriting the pointer with another value. Therefore the canary must either be a value the attacker does not know, or it must be a value the attacker can not easily write. One possibility is the usage of a random canary. The attacker could possibly write the flag, but since it is not known, the malicious user would first have to find and exploit a vulnerability that allows for memory-disclosure. There have been attacks in the past that used a combination of memory- or address-disclosure and memory-corruption to exploit a system. This proves that a random canary is no insurmountable obstacle for a determined attacker, but it is an additional hurdle to be jumped.

Another possible approach is to insert a flag that contains null-bytes or other 'forbidden' characters that cannot easily be written in the context of a buffer-overflow or some other memory-corruption attack (terminator canary). While semantic equivalences can be used to evade 'forbidden' characters in exploit-code, such characters can make an attacker's life difficult if they have to be written to a certain address verbatim. While an attacker could possibly overcome this obstacle, since the canary-value is well-known, it would stop simpler attacks.

A third algorithm for the creation of canary-values is by applying an XOR-operation to the protected address and a random value stored in memory. The advantage compared to random canaries is that the attacker has to read both the random 'seed' and the protected address in order to replace the canary with itself. This makes it more difficult to overcome the defense-mechanism, but still not impossible.

The main disadvantages of this defense-mechanism in general are that it is almost always

implemented as a compiler-patch and has to be activated manually. Additionally, a canary-flag offers no protection against attacks that allow the attacker to directly write to arbitrary addresses without overwriting other memory-regions. Such exploits are relatively rare, but they have occurred in the past and further vulnerabilites of this class may be found in the future. Much like with ASLR and non-executable memory, the insertion of canaries is not the ultimate protection against memory-corruption. As part of a combination of all possible countermeasures it would nontheless help harden a system against attacks.

2.6 Just-In-Time Heap Spraying

Many virtual machines that process interpreted code employ Just-in-Time Compilation. They thereby manage to combine the advantages of compiled machine code (fast execution) with the advantages of a virtual machine (no need to compile unnecessary code). Using a JIT-engine, the performance of an interpreter can be noticably increased. Given this performance boost, this technology perfectly fits for the highly contested area of browser development. Here, every instruction saved during the interpretation of javascript may lead to better results in various benchmarks, potentially leading to more favorable reviews and thus more users.

Unfortunately, the widespread use of JIT-compilation has also made it an attractive target for browser exploitation. So-called *JIT-spraying* is a powerful attack vector that was first described by Blazakis et al. in [5]. Using this method, an attacker can insert mostly user-controlled machine-instructions into the memory-space of the JIT-engine. In contrast to common code injection attacks, the executable code is typically inserted into the system not as a string, but as a series of XOR-instructions. Similar to most other small, simple instructions in JITed languages, an XOR-instruction is typically compiled directly into a native XOR operation as it is understood by the CPU. XOR EAX (opcode 0x35 in x86/x64) is a single byte operation with the target register EAX and a comparatively long (4 bytes) operand. Theoretically, attackers could use any instruction with short opcodes and relatively long operands, however, in practice, only XOR has been seen in the wild.

Usually, the operands of XOR are expected to be numbers, that is, plain user-data. However, CPUs without additional, virtualization-based features such as taint-tracking, are notorious for being unable to differentiate between data and code. Even these defense-mechanisms can be circumvented [7]. As a consequence, an attacker can encode the instructions to be executed as the operands of a series of XOR operations. A limiting factor is the XOR instruction itself: Operations have to be chosen so that the opcode 0x35, which cannot be modified by the attacker, does not cause execution to fail but becomes part of a semantic NOP. Since usually one byte of an integer operand is used to encompass the operand in a semantic NOP, three bytes per instruction remain for actual code. This makes the generation of an exploit somewhat tedious, but this technique can be used as the first stage of an exploit. For example, it can be utilized to load the actual payload into a nearby memory-segment, which, due to the use of JIT-compilation, is also likely to be mapped RWX. The only remaining step for a successful exploit is a vulnerability which can cause a misaligned jump into the memory containing the JIT-spray. After such a jump, there are three potential ways the code can be aligned:

- The CPU sees the code as a series of XORs, which, in this case, function as semantic NOPs.
- The CPU executes the code as intended by the attacker.
- The code is misaligned entirely, causing a failure due to SIGILL or SIGSEGV.

If a vulnerability that enables such a jump exists in the engine, the malicious script can then exploit the engine to execute the previously injected code, even if the most restrictive defense mechanisms, such as ASLR, non-executable memory and stack-cookies are in place. While disallowing RWX-accessible memory pages, as systems like SELinux do, does in fact protect from JIT-spraying, it will also break current JIT-engines in general, as they require memory pages with write and execute rights.

JIT-compiling the javascript-code `spray = (0x3C909090^ 0x3C909090^ 0x3C909090^ 0x3C909090)` would produce the executable code shown in listing 2.2. Executing this set of machine-instructions results in a value being moved to a register. The value is then repeatedly `XOR`-ed with a constant. If, however, there is a vulnerability through which an attacker can make the execution of the code start one byte after its actual beginning, the code that is executed would be as shown in listing 2.3. [32] The semantic of the code changes completely. Instead of a series of `XOR`-operations, a series of `NOP`s and comparisons is executed. As can be seen, the byte `0x3C` at the beginning of the `XOR`-ed constant serves to 'consume' the byte `0x35` introduced by the `XOR`-operation. The rest of the constant is payload which can be used to store malicious code.

Listing 2.2: Correctly aligned representation of compiled code

```
1 0x1A1A0100: B89090903C MOV EAX, 3C909090
2 0x1A1A0105: 359090903C XOR EAX, 3C909090
3 0x1A1A010A: 359090903C XOR EAX, 3C909090
4 0x1A1A010F: 359090903C XOR EAX, 3C909090
```

Listing 2.3: Incorrectly aligned representation of compiled code

```
1 0x1A1A0101: 90 NOP
2 0x1A1A0102: 90 NOP
3 0x1A1A0103: 90 NOP
4 0x1A1A0104: 3C 35 CMP AL, 35
5 0x1A1A0106: 90 NOP
6 0x1A1A0107: 90 NOP
7 0x1A1A0108: 90 NOP
8 0x1A1A0109: 3C 35 CMP AL, 35
```

2.7 Related Work

Apart from that, many protection mechanisms against common heap-spraying attacks fail against JIT-spraying due to the nature of the attack. Egele et al. proposed a defense against heap-spraying based on identifying shellcode in string buffers [14]. This mechanism cannot be applied

here since no string buffers are used to store shellcode. With NOZZLE [26], Ratanaworabhan et al. describe a general defense against heap-spraying that tries to detect NOP-Sleds on the heap and attempts to disassemble objects on the heap. The usefulness of this technique against JIT-spraying is limited, however, by the tendency of JIT-spraying attacks to use memory-leaks to find the jump targets. Apart from that, JIT-engines also tend to generate quite a large amount of data on the heap even under normal conditions. It may thus be possible to conduct a JIT-spray while keeping the heap-utilization beneath NOZZLE's threshold.

Bania [3] proposes a heuristic approach to defend against JIT-spraying, looking for a series of suspicious instructions in the disassembly of memory-regions that store JITed code. While this heuristic seems reliable, a systematic approach to mitigate JIT-spraying would be preferable. JITSEC [11] mitigates JIT-spraying by adding callsite-awareness to system calls implemented by the Linux kernel. While this method can protect against current forms of JIT-spraying, it is unfortunately operating system dependent. Furthermore, future attacks might combine JIT-spraying with Return Oriented Programming techniques, which are not affected by this defense-mechanism.

Ping Chen et al. [9] attempt to prevent JIT-spraying by selectively setting the execute-permissions for memory-pages containing JIT-code just before it is executed. This approach, however, ignores the fact that the execution of JIT-code may recursively re-enter the interpreter-function of a Just-In-Time compiler. In this case, memory would be executable during interpretation, which drastically reduces the gained security.

Rohlf et al. [28] discuss several ways of hardening JIT-engines. While techniques such as constant blinding, which applies an XOR with a secret key to all constants, or NOP-injection, which inserts random NOP instructions into the JIT-compiled code could prove to be effective, they can often be disabled either by exploiting a memory-leak in addition to the actual vulnerability or by spraying more malicious code.

Tao et al. [35] propose the insertion of code that is skipped if it is correctly aligned but triggers an interrupt if it is aligned incorrectly. The proposed defense-mechanism also employs the randomization of register-assignments and transforms the immediate operands of instructions. This defense-strategy would prevent the usage of longer sections malicious, JIT-code, but attackers could still use short snipplets of malicious JIT-code to gain control over the application.

Abadi et al. propose to enforce control-flow integrity through unique IDs that are checked at runtime [1]. However, the architecture depends either on data being non-executable, or the ID being unknown. Neither of these properties is ensured when JIT-spraying is combined with the exploitation of a memory-leak.

Even though *Lobotomy* is not the only technique that mitigates JIT-spraying, the approach proposed here aims to secure dynamic code generators against JIT-spraying while at the same time restoring the conceptional separation of data and code that was lost with the introduction of Just-in-Time compilation.

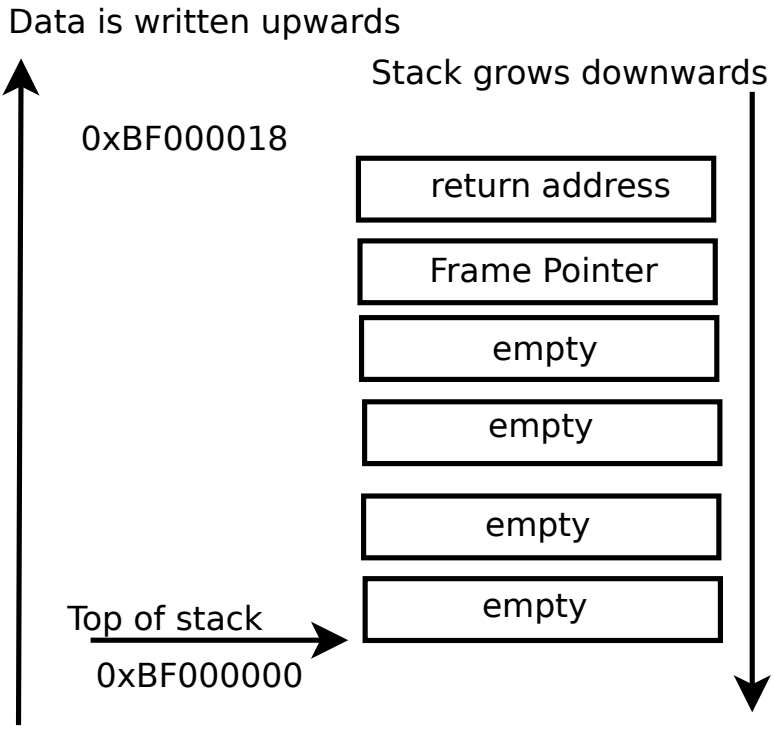

Figure 2.1: Stack before overflow

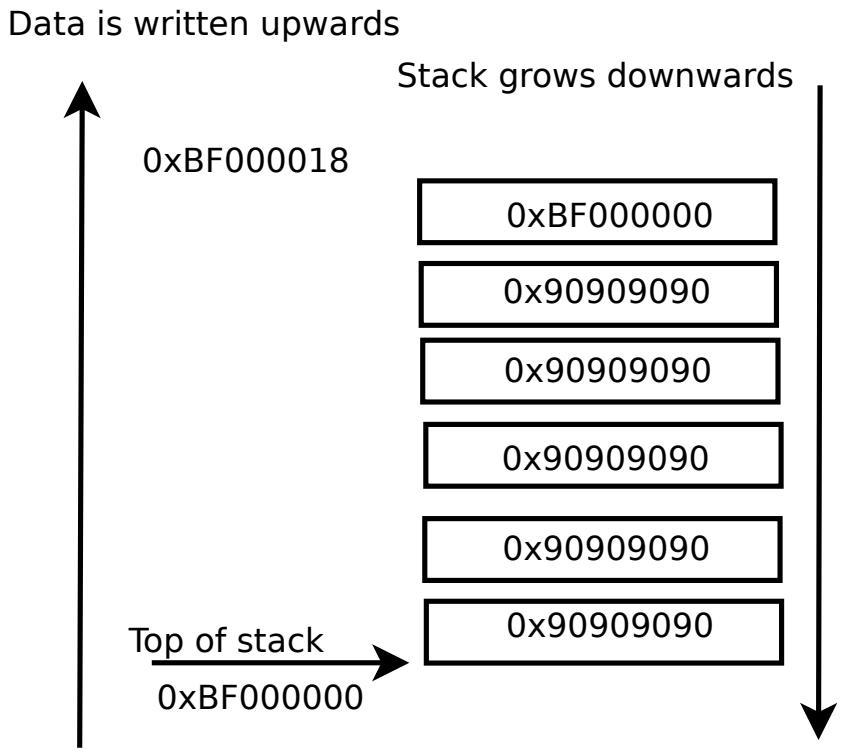

Figure 2.2: Stack after overflow

CHAPTER 3

Architecture

We set out to develop an architecture for Just-In-Time compilers of active content (such as Javascript or Flash) that is less vulnerable to JIT-spraying than the conventional architecture, in which security is sacrificed in favor of execution speed.

While speed is still an important goal for our design, our primary aim is to devise an architecture that is provably more secure when faced with a JIT-spraying attack than the baseline architecture. We thus need metrics to measure our success in this regard. Since JIT-spraying depends on jumping into an executable codepage, any page that is marked as executable is a potential target. Hence, one aim must be to minimize the amount of code through which an attacker might gain access to executable memory, i.e. the attack surface. We try to achieve this goal by keeping the number of lines of code run by the trace-executor as low as possible.

3.1 Architecture of TraceMonkey

As explained in [17], TraceMonkey, just like Tamarin in Adobe's Flashplayer, relies on the Nanojit JIT-engine for its JIT-compilation. Compilation and interpretation is controlled by a monitor and performed by `TraceRecorder`. The monitor keeps track of how many times a block of code has been processed so far. If the count reaches a threshold, a `TraceRecorder` is created and tasked with tracing the execution and compiling it into native code. 'Tracing' in this context means that while a script is being interpreted, its instructions are also stored by the trace recorder. Should the tracing fail, the monitor is informed. The latter keeps track of such failures and blacklists a piece of code if too many attempts at recording it fail. Once completed, the record is passed from the `TraceRecorder` to the underlying JIT-compiler, Nanojit in this case. The finished trace is then returned to the monitor. If the monitor encounters a program counter value that marks the start of a trace, it executes this trace instead of interpreting it yet again.

In order to execute a trace, memory is allocated for the execution state and the native stack. The execution state is the object through which information such as stack- and frame-pointers is

passed to the trace. The native stack contains values used by the trace. The monitor then calls the trace as a function, with the state as parameter.

When a trace terminates, it returns a structure called a `SideExit` which informs the monitor of the cause of the termination. In particular, it allows the monitor to keep a count of how many times a trace took a certain exit, such as a call to another piece of bytecode. If this count exceeds the above mentioned threshold, it too is now considered 'hot' and the monitor starts recording a new trace. The side exit is also used to keep the state of the interpreter and the native state synchronized.

While running a trace, potential exits are represented by so-called 'guards'. A guard is an assumption made at the time the trace is compiled. As long as it holds, the control-flow of the trace mirrors the control-flow during the recording. When the assumption is broken, the control-flow of the native execution is considered to have fallen off the trace, causing the compiled function to return. The monitor keeps track of these events by means of the side exits. If a new trace is recorded that represents the control-flow beyond a guard, a jump to this trace is patched in instead of that guard.

3.2 Hardening TraceMonkey against JIT-spraying

Based on the layout of the tracing engine, we can now try to isolate the trace execution from the trace recorder to prevent an attacker from executing code through a vulnerability in the recording- or monitoring-phase.

Besides mapping the codepages with read/write permissions in the compiler, we try to make JIT-spraying attacks even more difficult by mapping these critical memory regions read/execute to prevent an attacker from loading more code into the page that is currently being executed.

Thus, our first attempt was to simply modify the code allocator in Nanojit to allocate codepages from a shared memory file, mapped as RW. We added a `clone()`-call before `ExecuteTrace()`, the function containing the call to the trace. When this part of the engine is first run, `clone()` is called and a new process is created. It should be considered that initially, `fork()` was used to create a new process instead of the later `clone()`. We will return to this issue in Section 4. A separate section of shared memory was added to allow easy communication between the parent (the monitor) and the child (the executor). This shared memory object contains, among others, pointers to mutexes for synchronization, the state-object, the trace to be executed and the side exit taken after the trace completes. It also contains data-fields used for the management of shared memory and can be used by one process to pass boolean information to the other. In general, it serves as the centerpiece of the communication between compiler and executor and is henceforth referred to as 'communicator'. Figure 3.1 shows the components of the modified architecture. It differs from the original architecture primarily in that it is separated into different execution-contexts and that certain function-calls that cross between different components require a context-switch. Figure 3.2 shows the anatomy of a basic context-switch.

Upon creation, the newly spawned process immediately runs into a blocking semaphore. Two other semaphores are used to control concurrency between the executor and its parent, henceforth referred to as 'compiler'.

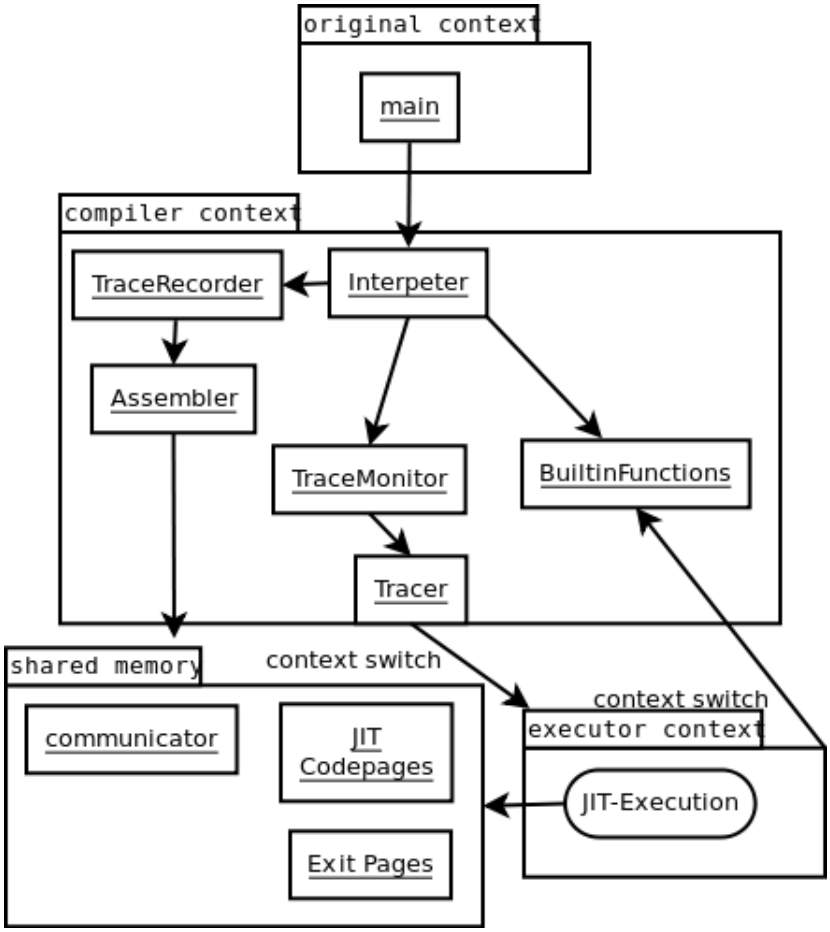

Figure 3.1: Component diagram

It would be possible to remove the strict ordering between compilation and execution to increase performance. Unfortunately, this would cause a further increase in complexity, which was deemed unnecessary for a proof-of-concept implementation.

After its creation, the executor-process initially waits on a semaphore. To trigger execution, the compiler stores status information about the trace to be executed in the shared memory segment reserved for this purpose. The executor's mutex is then signaled to, while the compiler waits on its own semaphore. This functionality is parceled into a function that can be used throughout the application to initiate a context-switch between the two processes. When a trace is to be executed, the final version of our architecture uses the `setcontext()`-system call to trigger a context-switch. Early versions used more straight-forward means such as a simple function-call.

The executor will then activate a context that was saved by the other process during the course of the context-switch. In the final version, this is simply done through another call to `setcontext()`. In initial versions, a more complex mechanism was used to transfer the con-

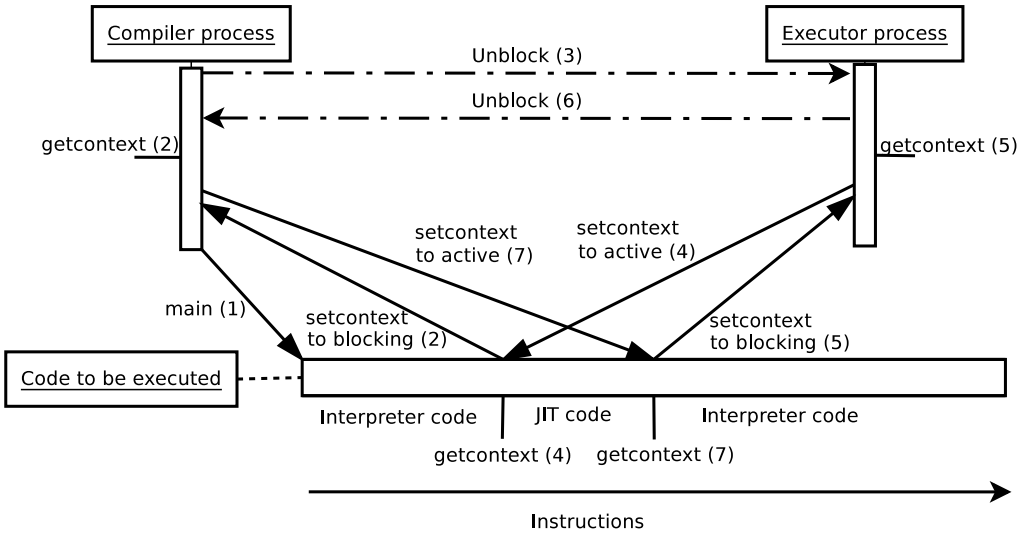

Figure 3.2: An exemplary context-switch to execute JIT-code

trol to the intended point. Next, the executor marks the code-pages of the trace as executable and finally executes the trace. It should be noted that successful execution of the trace depends not only on the trace itself, but on the code associated with the guards in place and the instructions responsible for leaving the trace in an orderly fashion. These short sections of code associated with each possible side exit from the trace need to be marked as executable as well, before the trace is started. After completion, the communicator is used to store the trace-information encapsulated in the TracerState object passed to the trace, as well as the exit taken to leave the trace. The executor then proceeds to unmarshal the data on the stack of the virtual machine back into the physical stack and updates several registers of the virtual machine. Finally, the executor saves its current context and triggers the context-switch to allow the compiler to resume before pausing itself.

Apart from the context-switch after the completion of the trace, additional context-switches were inserted for native builtin-functions which may be called from the trace. These context-switches operate in the same way as the normal case described above. However, the direction is reversed in that the executor first triggers the context-switch to restart the compiler, which activates the context left behind by the executor before the start of the builtin-function. After completion of the function, the compiler returns the control-flow to the executor. No memory is remapped for this context-switch. In the current version, some simple and very frequent builtin-functions are excluded from this treatment and still run in the context of the executor due to the slowdown that would otherwise occur. The context-switches for one function (`charAt()`) alone slowed down the application by 30%. Still, if required, the context-switch could be included in a facility of TraceMonkey responsible for handling all builtin-functions. This can be done relatively easily since the control-flow for all calls to builtin-functions runs through as set of macros in the file `builtins.h`.

Figure 3.3 shows a walk-through of an exemplary execution-sequence using the latest ver-

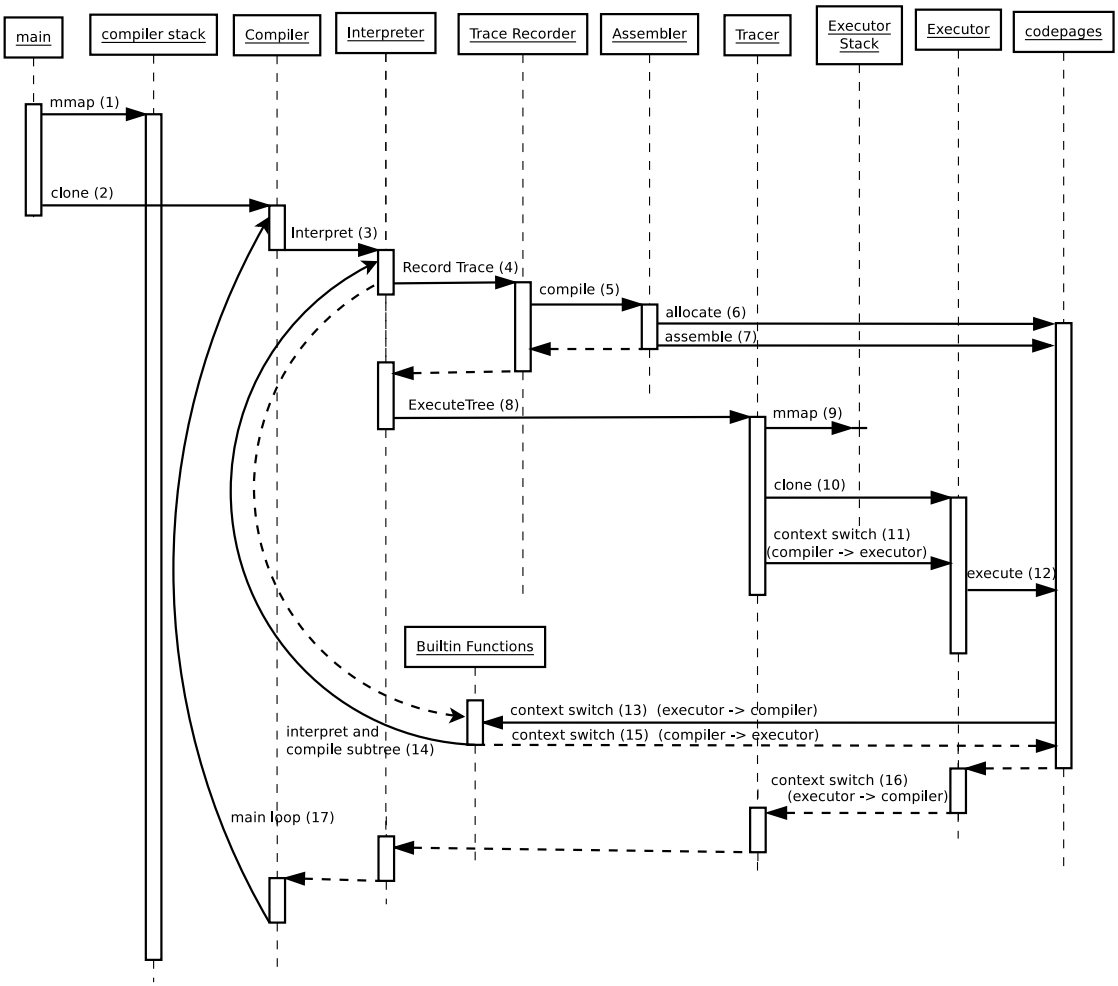

Figure 3.3: Sequence diagram

sion of the context-switching mechanism. The lifetime of the object `compiler stack` is used to indicate that computations of both processes takes place on the stack allocated for the compiler. The native stack which the main-function operates on is only used when a process waits on a semaphore after a context-switch. The basic execution-sequence of the engine has not been changed. However, the objects `compiler stack`, `compiler`, `executor stack` and `executor` were added. Correspondingly, calls (1) and (2) were introduced to generate the compiler stack and to start the compiler process. The process then follows the control-flow as specified by TraceMonkey until a trace is to be executed. At this point, the executor-process starts (10) and waits for the semaphore. Although it is not actually used, a stack for this process is allocated beforehand (9). Subsequently, a manual context-switch, involving a `post` to a semaphore and a `setcontext()` call (11) starts the executor. Now, the code compiled at runtime can be executed (12). In our implementation, this implies a call to `mprotect()` in order

to make the corresponding memory executable. A more optimized implementation could map these memory-segments into the executor-context with `execute` permissions in the course of (11) or even (10).

Running JIT-code might call builtin-functions (13). These builtin-functions make heavy use of the state of the `TraceRecorder` and the Interpreter and should be seen as belonging to the context of the compiler. For this reason, we introduce a context-switch from the executor to the compiler at this point. After the switch, the compiler calls the builtin-function requested by the executor. The builtin-function may in turn call back to the interpreter itself (14), which might cause the recursive compilation of another trace. Once the recursive call returns, control is handed back to the executor running the JIT-code (15). After completion of the JIT-code, we switch back to the compiler-context (16). From there on, the control-flow matches that of the unmodified engine.

3.3 Multi-Processing Challenges

Whenever two or more processes interact within a given system, the complexity of that system increases dramatically. The tracing engine of Firefox is not exempt from this rule. Thus, a number of problems have to be overcome in order to successfully implement the idea described above. First, it is necessary to make certain objects, such as the 'runtime'-object, which provides interfaces for memory-allocation and similar tasks, available to two distinct processes. According to its documentation, TraceMonkey [17] is not designed for this. Thus, we have to ensure that these objects are accessed in a deterministic manner and that the two processes are indistinguishable from the runtime-object's point of view. We also need to guarantee that all objects and variables that need to be shared by the two processes are shared. This obviously includes the memory region containing the compiled code and user-data. It is equally important though that the variables used by the JIT-engine to determine the control flow, such as leaving a trace or properties of a function that is being called, are shared as well. The same applies to variables used in various sanity-checks. By default, these conditions and invariants are implemented as integer-flags and reside on the stack.

Given these circumstances, an attempt to separate the executor from the compiler using normal processes as created by `fork()` is likely to fail: The two components will eventually lose sync because they do not share the same stack. In particular calls to a native C++ function from a trace often cause desynchronization. In the following a desynchronization likely leads to unexpected behavior and eventually to the violation of an assertion or even an outright crash.

While pointers to the codepages are fairly easy to localize and to handle, it is more difficult to modify the allocation of the control-flags. These flags are declared in header-files and used all across the JIT-engine. The engine expects these flags to behave and to interact in a certain manner, i.e. be located at certain offsets from each other. In fact, various sanity-checks are based on these interactions. Splitting the engine into two different processes may break these invariants and cause the associated checks to fail.

The situation is further complicated by the fact that once the executor-process is spawned, memory regions allocated by the parent are not automatically available in the child. In addition

it is not always possible to reproduce an allocation of a chunk of memory at exactly the same address as in another process.

Since memory management is a critical component of TraceMonkey and Firefox in general, any modifications can have far-reaching and unforeseen consequences and should be kept at a minimum. Nonetheless, we require a custom memory manager underneath the memory management functions provided by TraceMonkey to manage the file-backed shared memory, which we need to allow implicit communication between the two processes after the `clone()`.

It is important to note here that most of these issues are not the result of a flawed design, but side-effects of the attempt to modify an existing JIT-engine that was never designed to work as we intended. Instead, most considerations for the original TraceMonkey design are performance-driven. Still, we think that the value of a working modification of a well-known and functional engine outweighs the hassle encountered during implementation. Developing an entirely new javascript-engine as proof-of-concept for this design might have been easier. However, it would be very difficult to compare the runtime-behaviour of a newly developed engine with that of a highly optimized and refined virtual machine such as TraceMonkey. In particular, we would be hard-pressed to determine whether a slowdown is caused by the shortcomings of a newly designed engine and its lack of optimization, or if it is caused by the separation into two different processes.

Due to the issues mentioned above, we also explored an alternative approach similar to [9]. In this approach, the separation between compilation and execution is not achieved by additional processes, but merely carefully placed calls to `mprotect()`. However, evaluation of this approach showed that it is not suited for TraceMonkey. While the absence of `clone()` circumvents any desynchronization-issues and other problems that can arise from the decoupling of compilation and execution, the pages containing side exits and previously compiled code-fragments cause other difficulties. Either, the access-permissions of these pages are changed whenever the control-flow enters and leaves the execution-stage, or these pages are ignored and left with write- and execute-permissions. The first case leads to an increased performance-loss, while the second case leaves significant amounts (possibly megabytes, as shown in the evaluation of the NX-Approach in Section 5) of memory unprotected during and after trace-execution. This is caused by the fact that the access-rights to the pages in question need to be RWX since the compiler may have to write additional code there after trace-execution has completed, which is precisely the case we set out to avoid. Worse, traces may be recursively nested in that a trace calls a builtin-function, causing another trace to be compiled, which completely nullifies any security gained by this approach.

Another possible solution would be to create a new child process whenever a trace starts. This process is then responsible for the execution of the trace and is terminated again after the trace completes. Compared to the initial architecture with one process that is used for all traces, this design was fairly reliable since each trace started with a clean environment, influenced only by the state of the interpreter before the fork. If the interpreter-state is consistent, the state of the executor is also consistent even if the previous trace entered an inconsistent or corrupt state. However, two problems prevent this approach from being suitable in practice:

- Data allocated by one trace and not written back into a memory-location controlled by

the interpreter is lost after the completion of this trace. If trace-data needs to be retained for longer periods, additional modifications would have been necessary that would have caused unforseeable complications.

- The creation and destruction of this many child-processes causes a massive loss of performance. The constant allocation and deallocation of data-structures associated with the process itself, as well as the frequent destruction of various object-structures increases the runtime by a factor of 175 or more, as shown by the JS Trace Test Suite shipped with TraceMonkey. The precise reason for this crippling slowdown was discovered using the the coverage- and function-profiling tool `callgrind` that is shipped with the well-known debugging- and profiling-tool `valgrind`.

The initially conceived solution, at this point still using completely independent processes, was therefore reexamined in order to solve the issues that could hinder a successful implementation.

CHAPTER 4

Design and Implementation

Due to the complexity and high degree of optimization, it was decided to keep changes to the JIT-engine of Firefox as minimal and as local as possible. Thus, no interfaces between individual components were modified. However, some components had to be drastically modified in order to allow the engine to function when split into two parts. In particular, changes to the following elements are of consequence:

- The Memory Manager has to be modified to permit the allocation of file-backed shared memory so that the two processes may share data and implicitly communicate through their objects. This also requires a solid API between the processes that allows one process to map memory that was mapped by the other process in exactly the same location.
- A critical structure, containing much of the information that directs the control-flow of the tracer was moved to the shared heap-space.
- An explicit context-switch has to be introduced at critical points in the code to ensure that as much native C++ code as possible is executed by the compiler instead of the trace-executor.
- Finally, the usage of the structure `TracerState` has to be carefully examined. We have to ensure that the information stored therein is used consistently.

4.1 Memory Management

By default, `nanojit` assigns all memory-pages it allocates in order to store native code the access-rights `RWX`. Due to this policy, no calls to `mprotect()` have to be made after the initial allocation, which leads to a performance-gain. However, this behavior is obviously no suitable basis for an architecture meant to be resistant to JIT-spraying. Since the tracer needs to selectively set execute-permissions, the default access-rights were changed to `RW`. This still

allows the engine to write machine-code into newly allocated segments of memory, but that memory is not immediately a target for JIT-spraying.

Since it is necessary to access the trace from across process-boundries, the trace and all data it needs for successful execution need to reside in shared memory. Memory management in nanojit is handled by several allocators with different lifetimes and purposes, such as code, permanent data or temporary data. These allocators provide basic memory-management functionality such as `allocate` and `free` and work similar to the `malloc()`-family of functions in the LibC. With the exception of the `CodeAllocator`-class, all Allocators depend on a function implemented in the Tracer which wraps raw `malloc()`-, `calloc()`- or `realloc()`-calls and adds some book-keeping functionality. The CodeAllocator is exempt from this, being managed entirely by nanojit, while the other allocators are under the control of TraceMonkey. Ultimately, memory-allocation from the system to the Allocators happens in a set of utility-functions. These functions were modified to use a custom heap-management function which is backed by a file in the shared memory space of the Operating System. The `CodeAllocator` in nanojit was also adapted to call this custom heap manager. The heap-manager preallocates large sections of shared memory upon initialization and gives it out in smaller blocks when required. This increases the memory-footprint of TraceMonkey, but since TraceMonkey itself keeps large amounts of memory in reserve, rarely ever to be released, the effect is expected to be less and less noticable the longer the engine is running. It would also be relatively easy to change this behavior for more memory-critial and less time-critical applications. Preallocating memory also has the advantage that it is trivally shared when the Executor-process is spawned. Nontheless, there are cases where either the executor or the compiler has to allocate more memory, which has to be made available to the other process.

Thus, direct means are required for both processes to allocate an area of memory that has been previously allocated by the other process. To this end, the `mmap()` system-call is used in conjunction with the `MAP_FIXED`-flag, which allows the user to specify an exact base-address for the chunk of memory to be allocated. A facility called we called 'twosided memory' was implemented to replace `mmap()` in the tracing engine. When a process requests new memory, a chunk of memory is allocated from a shared-memory file. The base-address in which this chunk was mapped, as well as the size of the block and the offset in the memory-file is written to the communicator-object. The current process then initiates a context-switch after setting a flag requesting a memory-allocation from the other process. The other process, upon waking, reads the previously written information and passes it to its own `mmap()`-call, this time with a fixed address and offset in the shared-memory file. After the memory-allocation, another context-switch back to the original process takes place, which continues its execution. This 'twosided memory' serves as a replacement to the native `mmap()` calls that are occasionally used in TraceMonkey. Unfortunately, it is not suitable as replacement for the `malloc()` family of functions in LibC. It would be possible to build an entire memory allocation library to mirror its LibC counterpart, but the result would most likely not be stable enough and prone to memory-corruption.
We thus used an existing library called `CHeapManager`, which can, as mentioned above, be supplied with a pre-allocated chunk of shared memory.

Once compilation of a trace is completed, trace-execution starts. If no executor exists yet,

the process is spawned at this point. The code-pages of the current trace as well as any new side exits that may have been generated by the compiler are then made executable for this process alone. Once these instructions have been made executable, the list used to access them is cleared. It is not necessary to keep a list of previously executed fragments, since in the memory-space of the executor, the fragments stay executable during the executor's lifetime. As a result, the program does not have to iterate over, create or destroy large data-structures, leading to a noticably reduced runtime-overhead compared to that of the NX-approach. In scenarios where runtime is not an issue and maximum security is desired, it would be possible to remove the execute-permissions from the code-pages after the trace completes. This would add another layer of separation at the cost of slower trace-execution. However, it is questionable if much security would be gained from such a measure, since the code-pages would still be executable for the majority of the executor's runtime.

4.2 Nested Traces and DeepBail

One phenomenon that can occur during trace-execution and that needs particular attention is nesting of traces, or rather, the abortion of nested traces. Traces are considered nested when the loops that caused them to be generated are nested. This means that the call-graph of the trace consists of an outer and an inner graph, where the outer graph (or trace tree) calls the inner [4]. This condition does not usually interfere with the modifications made for Lobotomy. However, their abortion may cause complications in certain cases. As mentioned before, some utility-functions in the tracer are implemented directly in machine-code in order to increase performance. These native functions can then call functions such as js_Interpret, which eventually cause a new trace to be recorded. Conceptionally, such traces are one trace with a call to an opaque, native block. In practice though, the trace may abort with a condition called a Deep Bail. In such a case, the control-flow executes code associated with the compiler-process before returning from the trace, instead of directly jumping back to the exit-point of the executor. The consequence is that the roles of executor and compiler are no longer as strictly disjoint as they would conceptionally be. Since code-pages are mapped into the executor with read- and execute-permissions only, the compilation of this kind of trace could, at worst, crash. Additionally, this issue runs contrary to the design-goal of isolating the code and memory used by the executor from areas used by the compiler, as far as this is possible.

To solve this problem, context-switches were added to all functions that could cause nesting of traces and Deep Bails, in addition to the obviously necessary context-switches at the beginning and the end of the trace. Two options were considered to implement the context-switching. The first is a simple approach based solely on semaphores and an executor process that runs in a loop. The other option is based on `getcontext()`/`setcontext()` and was ultimately found to yield better results, but is also more complex and invasive.

In order to perform the context-switches, builtin-functions that can be called from the trace were replaced by a macro that has the same signature as the original function. This is facilitated by the fact that TraceMonkey itself uses macros to call these functions from the trace. The functions can therefore not have an arbitrary number of parameters, as there are only eight builtin-macros, each with a number of parameters from one to eight. The macros that were

introduced also take as argument a pointer to the function they replace. The original functions were renamed to `realfunction-name()` and are called as such in the macros. The context-switching macros first check if the executor-process is currently active. If so, a context-switch is performed from the executor to the compiler. The compiler then calls the renamed original function. The executor-process is reactivated after the function returns (regardless of whether it returns due to a Deep Bail or because of a successful completion. The executor returns from the function generated by the context-switching macro. It return to the trace and is then responsible for updating the status-object to signal a Deep Bail-condition. It also unmarshals the VM-stack and the virtual machine's registers and then switches its context with the compiler-process.

Listing 4.1 shows one of the context-switching macros. The trailing backslashes after each line were removed for legibility. The variable `shareptr` is the communicator-object responsible for communication between the two processes. The function `switchContextToWait()` encapsulates the logic of the context-switch itself. The second parameter of `switchContextToWait()` indicates the direction of the context-switch. A value of `false` switches from executor to compiler, `true` causes a switch in the other direction. The parameter `PTn` represent the types of the replaced function. The parameter `func` is the name of replaced function. `RT` is the function's return type while `LINKAGE` stands for optional modifiers such as static or extern. `MAYBEFAST` is a parameter that can hold the calling-convention used by the original function. In TraceMonkey, this may be `JS_FASTCALL` or nothing (empty parameter).

Listing 4.1: Macro definition of context-switching function

```
1 #define GEN_CONTEXTSWITCHER4(func, PT0, PT1, PT2, PT3, RT,
2 LINKAGE, MAYBEFAST)
3 LINKAGE RT MAYBEFAST func(PT0 p0, PT1 p1, PT2 p2, PT3 p3) {
4     RT ret;
5     if((shareptr != 0) && shareptr->childActive){
6         switchContextToWait(shareptr, false);
7         if(!shareptr->isExecutor){
8             shareptr->childActive = false;
9             ret = real##func(p0, p1, p2, p3);
10            shareptr->childActive = true;
11        }
12        switchContextToWait(shareptr, true);
13        return ret;
14    }else{
15        return real##func(p0, p1, p2, p3);
16    }
17 }
```

This condition is also one of the major arguments for a context-switch based on `getcontext()`/`setcontext()`. Interrupting and resuming execution of both the executor and the compiler so as to keep their control-flow synchronized is a very complex endeavour with context-

switching based on function-calls. Unfortunately, the location to where a function-call from the compiler to a Deep-bailing function would return to is not the point where the execution of the process should resume after the trace finishes. The natural solution is the use of `setjump()`/`longjump()` and their descendents, `getcontext()`/`setcontext()`. At this point though, it was concluded that the entire logic of the context-switch could be implemented much easier in this fashion. On the other hand, context-switching that is entirely based on `getcontext()`/`setcontext()` requires that each process is able to operate on the stack of the other process. This is commonly thought of as troublesome, and the manpage of `clone()` insists on the allocation of one stack per process.

4.3 The ThreadData object

Even with the compiler isolated from the executor to limit the execution of code in the wrong process, the fact that at least two context switches occur per trace between two processes that share some, but not all of their memory would causes significant amounts of instability. It was thus attempted to identify a point where the data required by both processes is available to be shared between them. For this purposes, an object originally on the general purpose heap was identified. This ThreadData object holds much of the status-information of the tracer as well as data required by the javascript virtual machine, such as pointers to the base of the stack, the currently active javascript compartment and the stack itself. The init-function of the tracer's runtime object was modified to initialize the variable holding the ThreadData object with shared memory.

Additionally, several other data-structures are implicitly shared, because they are allocated using the memory-allocation utilities of the javascript-engine. As these built-in allocators were modified to return memory backed by a shm-file, structures and flow-control data allocated in this way are always available to both processes. Nontheless, some objects and structures must reside on the stack, either since the JIT-compiled code expects it to be there or because the position of an object in memory is so closely tied to other parts of TracerMonkey through assertions, offsets and 'magic values' that they cannot be moved easily. Thus, it would be advantageous if data located on the stack could be shared as well.

Fortunately, the fact that a process immediately blocks itself after waking another allows for the two processes to share their stacks.

4.4 Shared stacks and context-switching

Both the compiler and the executor must have seperate stacks. However, the `clone()` allows the user to provide any suitably aligned and sized piece of memory for use as stack by the child-entity, even if it is drawn from a shared-memory file. To replace the physical machine's native stack with shared memory, it is necessary to pre-allocate two large blocks of data immedately after the application starts, so that no important information is stored on the native stack, which will only be used when a process intends to wait on a semaphore. After stack-allocation, TraceMonkey must be made aware of the change of the base of the stack, since this information is required to detect over-recursion and similar conditions, as well as to accurately calculate various

offsets. Once the engine is aware of the change, the compiler proceeds to execute as usual. When a trace is to be executed, a new process will be `clone()`-ed, which is passed the other chunk of pre-allocated memory to be used as stack. This memory will, however, see no true use. The compiler, in the course of the context-switch, saves its context, which includes the compiler's stack and then jumps to a context on the native machine-stack. The compiler's destination-context is located on the native stack because this is where the first `getcontext()`-call was made. After the compiler signals to the executor's semaphore, the executor switches its own context with the one just saved by the compiler, from where it continues execution. Listing 4.2 shows the code that implements this behaviour. After the trace is complete or when a builtin-function or a request for memory-allocation is encountered, the same process takes place once more in the other direction, causing the compiler to wake up, leave its current context on the native machine-stack behind and to take up the context left by the executor.

Listing 4.2: The function that triggers the context-switch

```
1 void switchContextToWait(shared *shareptr, bool fromCompilerToExecutor){
2     getcontext(shareptr->executorContext);
3     bool checkflag = shareptr->isExecutor;
4     if(fromCompilerToExecutor){
5         checkflag = !checkflag;
6     }
7     if(checkflag){
8         shareptr->isExecutor = fromCompilerToExecutor;
9         setcontext(shareptr->executorWaitContext);
10    }
11 }
```

The field `executorWaitContext` of the communicator object `shareptr` represents the destination-context, where the process that is being switched out will wait. The field `executorContext` represents the context into which the process that is being switched in will take over. The variable `checkflag` ensures that after a process switches in, it does not immediately switch out again. Note that no calls to `sem_wait()` or `sem_post()` occur here. The concurrency-control is the responsibility of the destination-context. This context is created by a function that causes the process that enters it to release the other process from the semaphore just before it enters its own semaphore.

Curiously, we discovered that the major difficulty in this architecture is not related to the use of `setcontext()` and `getcontext()` itself. It was not necessary to modify any part of the code itself in order for it to function after `setcontext()` transfers control from one point of the code to another, even when the instructions that wrote the destination-context to the communicator-object were executed by another process. The context stored by `getcontext()` is complete. Therefore, the executing process makes no difference. The main difficulties arose from the ordering of the two processes. The order in which the processes are freed from their semaphores and in which they execute after a context-switch has to be clearly defined and strictly maintained. Otherwise code is executed by the wrong process, which may reduce the security of

the architecture. More severely, a disorderly execution-sequence could cause executor and compiler to change place entirely. Since each context-switch has a clearly defined direction, such swaps will eventually cause deadlocks which are hard to debug. It is, after all, not always easy to find the exact reason for such a swap. Usually, the original cause was an unexpected scheduling of the two processes by the operating-system, causing a previously unforeseen execution-sequence. As a result, more semaphores were introduced than would be strictly necessary were the ordering less critical. There is once semaphore for each process as well as an additional semaphore that guards the context-switch itself. There is also a fourth semaphore that ensures a particular order of execution when the child-process is first created. In particular, we need to ensure that the parent process does not try to initialize a context-switch before the child-process has been run and has entered its semaphore. The use of this many semaphores may seem excessive. It may be that one or even two of the semaphores could be removed by more careful semaphore-placement and a different algorithm at process-creation. However, for this proof-of-concept implementation, we decided to err on the side of caution instead of trying to implement the context-switch in the most optimized way possible. Listing 4.3 shows the function that implements the sequencing-logic and contains the semaphores in use.

Listing 4.3: Concurrency- and sequencing-manager for context-switching

```
1  void getExecutorWaitContext(){
2      if(!shareptr->doExecWait)
3          getcontext(shareptr->executorWaitContext);
4
5      //We are the child, and we are first, so we should be alright
6      if(shareptr->childPid == 0) shareptr->childPid = getpid();
7
8      if(shareptr->firstWait && (getpid() != shareptr->childPid)){
9          //We are first and we are not the child, so we wait
10         waitForSem(&shareptr->mutex_execFirst);
11     }
12
13     if(shareptr->doExecWait){
14         if(shareptr->waitFlipFlop){
15             shareptr->waitFlipFlop = false;
16             if(!shareptr->firstWait){
17                 signalToSem(&shareptr->mutex_compile);
18             }else{
19         //We should be the child, and are first, so we signal the parent.
20                 signalToSem(&shareptr->mutex_execFirst);
21                 shareptr->firstWait = false;
22             }
23             waitForSem(&shareptr->sem_ctxt);
24         }else{
25             shareptr->waitFlipFlop = true;
26             signalToSem(&shareptr->sem_ctxt);
```

```
27                  waitForSem(&shareptr->mutex_compile);
28
29              }
30              if(shareptr->doRun == false) return;
31
32              setcontext(shareptr->executorContext);
33          }else{
34              shareptr->doExecWait = true;
35          }
36      }
```

The listing above shows the code that implements mutual exclusion and that controls the concurrency of the two active processes. If the flag `doExecWait` is not set, `getcontext()` is used to store the context to which the process will return to after a call to `switchContextToWait()`. After that, we determine whether the function is called by the executor (the child) or the compiler (the parent). If `childPid` in the communicator-object is not yet set, we set it to the current process-id. This means that the executor is the first to enter this function and will soon be ready to execute a trace if called to do so.

If this is the first call of the function (`firstWait` is set) and the stored PID differs from the id of the calling process, we assume that the caller is the compiler. It will then wait here until the child-process is actually ready to execute a trace. Once the child enters the function and `doExecWait` is set, it will enter the main body of the function. This block is controlled mostly by the somewhat aptly named `waitFlipFlop`-flag. This flag changes state whenever a process enters this block. Given our assumption that the child-process will always enter it first, the flip-flop variable will be set to true in the beginning. If the child-process then notices that it is the first time the function has been called, it will release its parent, the compiler from the semaphore it has been waiting in. It will immediately afterwards block itself in the semaphore `sem_ctxt`. If the executor is preempted by the newly released compiler, the executor will enter the blocking semaphore after it regains the processor. If the compiler has signalled to the waiting semaphore and put itself to sleep in the meantime, the child will simply step over the semaphore and continue its operations. Regardless of whether it is by choice or by scheduler, the executor is blocked at this point. Also regardless of the order in which the two processes first entered the function, the compiler will now enter the main body of this function. This is due to the fact that if this is not the first time the function has been called, the executor will signal to the semaphore meant for the compiler to wait in (`mutex_compile`). As the flip-flop flag has changed its state, it now expects the compiler to enter. The compiler will signal to `sem_ctxt` and will in turn wait on the above mentioned semaphore that is reserved for the compiler (`mutex_compile`). In case this is not the first time the function was called by the compiler, it may now continue if the executor has called a `signalToSem()` on that semaphore. If it has not, it will block until the executor arrives. If it *is* the first time the compiler entered this function, it will block and the executor will leave it instead. Regardless of which process leaves the main block of the function, it will typically do so through a call to `setcontext()` on the context stored in `shareptr->executorContext`, which is a misnomer, since it currently is used by both processes. The only situation a process will leave this function through 'regular' means is in

case of a shutdown of the application. This scenario occurs when `shareptr->doRun` is set to false. The sequence in the if-block under `shareptr->isAllocRequest` is currently unused. It is a remainder of the original, function-call based context-switching, in which a set of flags was used for a process to determine where to jump to. In the current state of the architecture, the location from where a process initiated a context-switch does not matter. Figure 4.1 illustrates the control-flow through this function.

The changes described above are the corner-stones of the modified architecture presented in this thesis. However, some other, comparatively minor but nontheless critical changes were necessary for the system to function.

Besides the implied allocation of various shared objects such as the communicator referenced throughout the code as `shareptr` and the memory-manager responsible for allocating file-backed shared memory, it is also necessary to inform TraceMonkey about the change of the process-stack. We used the depricated function `JS_SetThreadStackLimit` and modified it to suit our needs. As is, the function sets the stack-limit of the engine's current context to a given address which the stack must not exceed. This is used to implement checks against over-recursion and similarly undesirable states. We modified this function so that it also sets the field `nativeStackBase` of the `ThreadData` object. As mentioned above, this object holds data critical for the operation of the VM, such as the base of the native stack. This address is used, among other things, to marshal and unmarshal data before and after the execution of a trace. This marshalling is of great importance. The javascript virtual-machine implemented in TraceMonkey operates on its own 'virtual' stack, which is implemented as a userland-variable. This variable is a pointer to a memory-region allocated by mmap. We modified this allocation by adding the `MAP_SHARED`-flag. This allows the newly allocated memory to be shared between processes. However, this alone is not enough to restore the functionality of the engine after separating it into two processes. The reason for that is the above-mentioned marshalling of VM-data onto the native stack. This process can be seen as part of the manifestation of the virtual machine into code and data that can be executed on the physical machine. While the compiler itself (nanojit) manifests the code to be executed, the unmarshalling (in functions such as `ValueToNative()`) can be seen as the manifestation of the VM data. This process is highly dependent on knowledge about the physical stack. If this information (kept in `ThreadData->nativeStackBase`) is outdated, the data of the VM gets copied to the wrong address. To prevent such mis-writes, the location of the stack-base has to be updated as soon as possible. We do this in the main-function of the binary from which the tracer is started, immediately after the creation of the execution-context.

4.5 The state object

Another object of crucial importance to the tracer is the `state`-object. It is a structure of the type `TracerState` and holds many values that are critical for the execution and management of JIT-code. It is created by the compiler before the start of a trace and is allocated on the machine-stack. Due to the shared nature of the physical stack, this implicitly allows for the executor to access the object. However, simple availability on the trace is not sufficient. We need to ensure that the state-object is actually used on the trace. For this purpose, the object

has to be passed to the trace by reference. In an early version of the modified architecture, the state-object was passed by value for the sake of simplicity. This caused a mirror-object to be created which was not synchronized with the original state created before the start of the trace. Once the trace terminated, any changes made to the mirror-object were lost and the compiler was not informed of various events on the trace. However, we also have to make sure that the correct addresses and offsets are compiled into the trace.

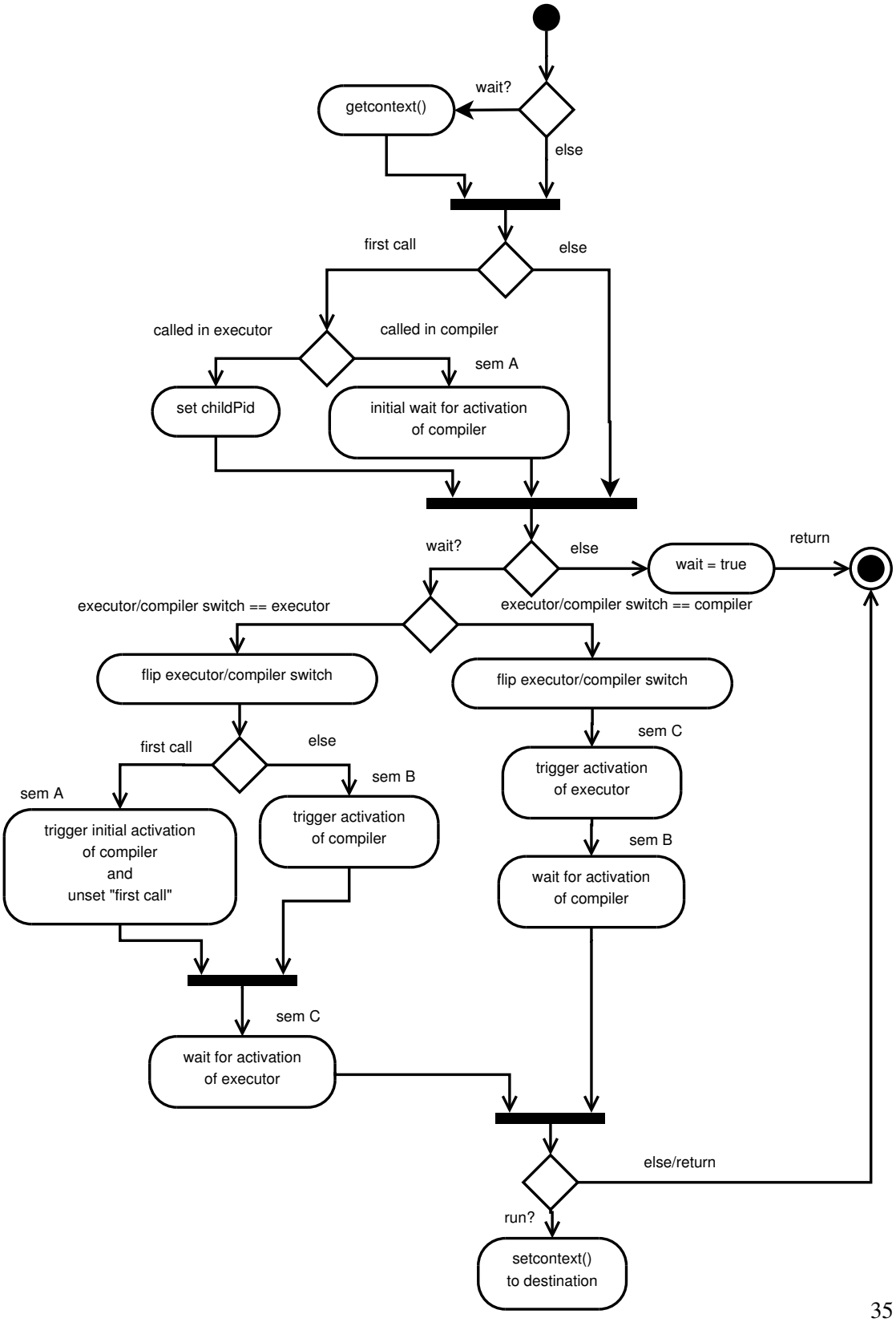

Figure 4.1: Control flow diagram

CHAPTER 5

Evaluation

The evaluation presented in this section covers both approaches discussed in the previous chapters. We first discuss the NX-approach, followed by the fully separated architecture and a comparison of the two designs. The unmodified version of TraceMonkey shall serve as baseline. Evaluation-criteria are the performance of the benchmarking and regression-testing tool shipped with Firefox when testing the different architectures as well as the extent of code that is run with execute-privileges.

5.1 NX-Approach

The NX-architecture was developed to evade the problems brought on by isolating an individual component of an existing, highly optimized system such as a javascript engine and running it as a process of its own. As such, it is fairly simple and the control-flow rarely deviates from the control-flow of the unmodified version. On the other hand, there is no hard separation between trace-execution and trace-compilation.

Performance

TraceMonkey is highly optimized to increase its performance and the modifications made for the NX-architecture do not undo these optimizations. Hence, we expected no dramatic loss of performance. However, we insert at least one additional system-call – an `mprotect()` to add execute-permission to the code-pages containing the trace. In practice, several `mprotect()`-calls may be necessary since the pages containing the side-exits have to be marked as executable as well. Since TraceMonkey avoids system-calls whenever possible, adding these extra calls will result in some performance overhead. This overhead was expected to be most noticeable in short traces that perform simple computations, or with code that often starts new traces. Its impact was expected to dwindle with increasing complexity of the traced code itself and with increasing duration of each trace. Another source of overhead is, as mentioned before, the management of

Version	Number of test cases	Percentage failed	Runtime overhead
Re-protecting NX	1087	3‰	9.04x
Non-re-protecting NX	1087	6‰	1.02x
Baseline	1087	0‰	1.00x

Table 5.1: Results of running the test cases with the two NX-versions and the unmodified baseline.

and iteration through the lists that are used to store previously used code-fragments and side-exits. In fact, an analysis using valgrind [22] and its plugin `callgrind` showed this to be the main source of performance-overhead in the NX-architecture.

We evaluated the performance of the NX-architecture by running the test cases of the regression-testing and benchmarking suite of TraceMonkey [16]. The corresponding Results are shown in Table 5.1. With re-protecting NX, the access-rights to side-exits and previously compiled fragments are set and reset before and after a trace. Consequently, non-re-protecting NX has the same setup, but this time, compiled fragments and side-exits remained executable after the termination of the trace.

A comparison of these two versions shows clearly that in the non-reprotecting variant, the overhead is reduced dramatically. Note that this reduction of runtime-overhead is not so much due to the reduced number of `mprotect()`-calls, but much rather due to not iterating over various lists of code-sections. Unfortunately, the fragments that remain executable during the lifetime of the tracer cause an equally drastic increase in terms of attack-surface. Finally, we also provide numbers on the unmodified version of TraceMonkey as a baseline.

Also notable is that seven test cases of the non-re-protecting and three test cases of the re-protecting variant fail. Such failures occur when the settings of access-rights to memory does not match the assumptions made by the engine. On the one hand, three test cases fail because the executor tries to execute code-fragments that are still protected `RW`. In the non-re-protecting variant, four additional test cases fail because the compiler tries to patch code that is protected `RX`.

Attack Surface

The primary goal of our system was to reduce the attack-surface in terms of executable memory pages. On the test-system, the pagesize is 4096 bytes. To evaluate our success in this regard, we compiled the non-reprotecting variant of the tracer into Firefox. We then used this modified browser in two ways:

- We visited a series of popular websites – the top one hundred given by Alexa [2].
- We simulated a normal user session by automatically interacting with common websites.

To automate the process, we leveraged Selenium [18].

Results of the first experiment can be seen in Figure 5.1. The graph is characterized by a more or less linear increase, with some plateaus after new memory is allocated. Occasional,

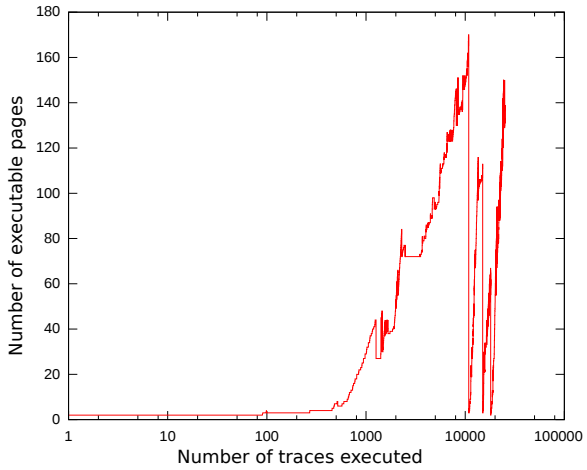

Figure 5.1: Executable pages (Alexa's top 100)

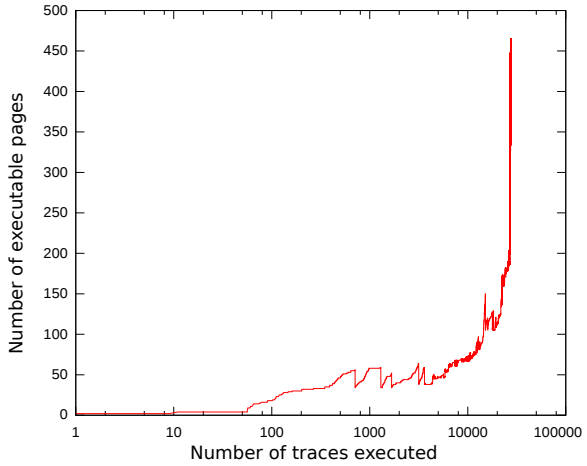

Figure 5.2: Executable pages during simulation of normal use.

small drops were caused by garbage collection events. The non-monotonous behavior at the end is due to the re-opening of the browser-window after the test and is not part of the actual test-run. It is, however, interesting to see that after the executable codepages cleared at the end of the actual test, subsequent trace-executions quickly brought up the number of pages to a comparable level.

For the second experiment, several common websites, such as `google.com`, `facebook.com`, `twitter.com`, `youtube.com` and a local news-portal were visited. For `google.com` we issued a simple query. On each site, the mouse was hovered over each active element, including links, images with alt-texts and other components that were found. Apart from that, the right mouse-button was pressed on each site and the window was, if possible, scrolled.

Therefore, although the set of sample-sites is far smaller here, the number of executed traces is similar up to this point. Finally, the site `maps.google.com` was visited. The map was moved in each direction and zoomed afterwards. The resulting graph in Figure 5.2 indicates a similar behavior as in the previous experiment, except for the sharp increase in allocated pages at the end, when the interactive map of `maps.google.com` was used. This shows that while usually the amount of executable memory is in the region of hundreds of kilobytes, it can also spike even for benign sites.

When hosted on a malicious website, a potential exploit would therefore still find sufficient executable memory to succeed. To avoid triggering heap-protectors such as `NOZZLE` [26], the attacker could cause the executable memory to build up at a rate similar to benign applications. The buildup might be disguised as a game or other feature which is likely to keep the user amused for long enough to store megabytes of potentially executable code in memory. Additionally, many features of the user-interface of Firefox are implemented in javascript. While it is difficult for an attacker to control javascript-code run in the context of the user-interface, vulnerabilities that permit access to this context have been found in the past. Even if an attacker does not control the code in the UI-context, code-pages generated for this context could be used as padding that forces the engine to allocate additional memory elsewhere.

5.2 Full Separation

In search of a conceptionally sound and robust way of hardening a Just-In-Time compiler against JIT-spraying, we conceived the idea of splitting the program into separate processes. Such an architecture bears the advantage of two independent virtual memory segments, creating a boundary between code-creation and code-execution that cannot be bridged in case of an attack. The security gain is based on the assumption that the process that creates and compiles the JIT-code runs significantly more, potentially vulnerable, native code than the process that executes the JIT-code. On the other hand, the fact that the engine consists of two processes that need to be synchronized is expected to cause an intrinsic performance-overhead. Also, any allocation of shared memory after the `clone()` needs to be performed twice.

Performance

While the performance of the NX-approach suffers because of the `mprotect()`-calls, we expect the overhead of the fully seperated architecture to be caused by context-switching and the duplicated memory-allocations. However, the evaluation of the runtime-overhead showed rather surprising results.

As can be seen in Table 5.2, full separation comes at a price of not quite five times the execution time for the test cases compared to the unmodified baseline with the JIT-engine enabled. With JIT compilation disabled in the baseline, the overhead is reduced to a factor of 1.87. We believe that by optimizing the modifications and more careful memory-management to avoid unnecessarily duplicated memory-allocations, it would be possible eliminate the cases in which the worst performance overhead occurs.

Version	Number of test cases	Percentage successful	Mean runtime overhead	95% quantile
Full separation	1087	98.2%	4.87x	1.27x
Baseline w/ JIT	1087	100%	1.00x	1.00x
Baseline w/o JIT	1087	100%	2.60x	6.86x

Table 5.2: Runtime of the fully separated architecture and the unmodified baseline

Since some test cases caused extreme overhead, we also compared the 95% quantile of the fully separated version with the same test cases in the two baseline variants. Results show that the reduction in runtime-performance for the vast majority of test cases is much less severe: they were only 1.27 times slower on average than the baseline with JIT and 5.41 times faster than the baseline without JIT.

Worst-case runtime overhead

Initially, our suspicions about the root-cause of the dramatic worst-case overhead (in one case over twenty times) of the modified architecture versus the baseline fell on the context-switching infrastructure. However, manual analysis of some of the worst-performing testcases disproved this theory. Rather, we discovered that there is an issue that causes superfluous memory-segments (internally called 'arenas') to be allocated. The engine seems unable to use previously allocated arenas. In search for a fitting storage-place for user-data, the engine iterates over a list of potential candidate arenas yet fails to find one. Ultimately, this causes the allocation of more memory which is then added to the list. This, in turn increases the time for the next search for a suitable storage-location, which also fails. In test-cases that often create new array- or object-fields, the time needed to iterate over a list of thousands of empty arenas adds up to a severe loss of performance. An engine that is to see productive use must, naturally, take greater care to correctly allocate and manage memory-segments than our current prototype does.

Listing 5.1: Partial listing of memory-segments allocated by the executor-process

```
1  ...
2  <shared libraries>
3  <code segements>
4  084e0000       12 rw---- [ anon ]
5  091cd000      132 rw----     [ anon ]
6  9c200000     1024 rw-s-  shm_cmap_common
7  <three more>
8  9c360000      132 rw-s-  shm_calloc
9  9c381000     3868 -w-s-  shm_calloc
10 9c748000     4000 -w-s-  shm_dalloc
11 <two more>
12 9d300000     1024 rw-s-  zero (deleted)
13 9d426000        4 rw-s-  zero (deleted)
14 9d427000        4 rw-s-  zero (deleted)
```

```
15 9d429000      252 rw-s-  zero (deleted)
16 9d468000        4 rw---    [ anon ]
17 9d469000     4096 rw-s-  zero (deleted)
18 9d869000     4000 -w-s-  shm_calloc
19 9dc51000     4000 -w-s-  shm_dalloc
20 <two more>
21 9e809000        4 -----    [ anon ]
22 9e80a000     8192 rw---    [ anon ]
23 9f00a000   400000 -w-s-  seperation_shm_allocHeap100
24 <LC_* locales>
25 b7818000        4 rw---    [ anon ]
26 b7833000        4 rw-s-  zero (deleted)
27 b7834000        4 rw-s-  zero (deleted)
28 b7835000        4 rw-s-  zero (deleted)
29 b7837000      252 rw-s-  zero (deleted)
30 b7876000       16 rw---    [ anon ]
31 b787a000        4 rw-s-  zero (deleted)
32 b787b000        4 rw-s-  zero (deleted)
33 <51 more>
34 b7894000        8 rw---    [ anon ]
35 bfd58000       84 rw---    [ stack ]
```

Listing 5.1 shows part of the output of the command-line tool `pmap(1)`. This program shows, for a given process-id, the address-ranges that contain valid, allocated memory, as well as the attributes of that memory (whether it is shared memory, the permissions that are set, the size of the allocation and similar characteristics). After a single execution of the testcase `testAtomize()` (see below for more details), there were a total of 62 segements consisting of four kilobytes (a single page) of anonymous, zero-filled shared memory. Given that each arena is 40 bytes in size, this equals 6200 superfluous arenas. The unmodified architecture has allocated 16 such anonymous segments after executing the same testcase, none of them zero-filled and all having a size greater than four kilobytes. This supports the assumption that there is a bug in the handling and allocation of shared memory in our modified architecture. Unfortunately, due to the complexity of the matter and for time-constrait reasons, we were not able to conclusively pinpoint the root-cause of the problem.

Errors

In total, 18 testcases fail when run by the automated testing framework. Detailed manual analysis has shown that the errors are caused by several classes of bugs in our solution. While these bugs are caused by the complexity of the implementation, they are not intrinsic to the architecture itself. An engine developed with this architecture in mind could likely handle much of the complexity of our solution implicitly.

The bugs that remain in the current implementation can be classified as follows:

- Errors that occur because of a mismatch between trace and statically compiled code. These errors may manifest as assertion-failures (when the statically compiled code behaved in an unexpected way when marshalling or unmarshalling trace-data), or as segmentation faults (when the trace behaved unexpectedly).

- Overrecursion bugs that occur when a testcase runs out of stackspace. Some highly recursive test cases may run out of stack-space because the stack is allocated when the process is created. In an implementation suitable for a production environment, the engine would have to manage its own stack. It would be possible to dynamically increase the size of the stack using `mremap()`. Unfortunately, this function can fail if it is unable to extend a given area of memory. In such a case, a parameter can be supplied which allows the Operating System to move the contents of the memory to another, larger segment. An implementation that uses `mremap()` must therefore ensure that the execution-environment is not corrupted by moving data to another location in memory. We believe that ensuring this would not be a fundamental architectural problem, but it would be a significant engineering-challenge that was considered to be out of the scope of the proof-of-concept implementation. It should be noted that in these testcases, the recursive behaviour is the same in the baseline (the error is not caused by misrouted control-flow.)

- A fragment of dynamically compiled code that is being called in a trace is not executable. This condition is presumably caused by the loss of a pointer to a code-fragment.

Table 5.3 shows the underlying error for each failed testcase. Segmentation faults are centered around the end of the trace, with the exception of the failure of `check-deltablue`, which failed at the generation of the trace, and is more likely associated with the other subclass of trace-related errors. The root-cause of these errors is that the trace behaved in an unexpected way. Either, the control-flow did not leave the trace where expected or a variable was written to an invalid or unexpected location. Additionally, the assertion that stopped `check-crypto` is triggered right before the trace completes (a copy of the last instruction-pointer is written to a wrong address).

Most other assertions are triggered before the trace is started or after a recursion is started. (likely including the failure of `check-date-format-xparb.js`) These errors apparently indicate that the statically compiled code attempted to pass parameters to the trace in an incorrect or unexpected way.

The last failed testcase in Table 5.3 is unusual insofar as it is terminated by a user-level assertion. However, the nature of the assertion (got 197, expected 198) hints that the root-cause may be a premature end of the trace. This puts it in relation with the class of errors caused by unexpected behaviour on the trace.

In general, debugging errors relating to the generation, calling and management of dynamically generated code is difficult, especially in a system as complex as TraceMonkey.

Code Coverage

To prove the security gain of the fully separated architecture compared to the baseline-architecture and the NX-variant, it is necessary to evaluate the ratio of code executed by the executor-process

Testcase	Reason for failure
bug617139.js	overrecursion
testFewerGlobalsInInnerTree.js	Assertion failure: !cx->iterValue.isMagic(JS_NO_ITER_VALUE), at jsiter.cpp:1043
testWatchRecursion.js	overrecursion
t017.js	Assertion failure: *(uint32 *)slot != 0, at jsvalue.h:209
bug554651.js	overrecursion
bug557070.js	overrecursion
bug577705.js	Assertion failure: *(uint32 *)slot != 0, at jsvalue.h:209
check-3d-cube.js	Segmentation fault in `LeaveTree()` due to misrouted control-flow
check-3d-raytrace.js	non-executable fragment
check-access-fannkuch.js	Infinite run, either due to indefinite slowdown (see performance-evaluation) or misrouted control-flow (infinite loop)
check-access-nbody.js	Segmentation fault in `LeaveTree()` due to misrouted control-flow
check-date-format-tofte.js	Segmentation fault in `LeaveTree()` due to misrouted control-flow
check-date-format-xparb.js	Segmentation fault in `GetPrimitiveThis()`
check-crypto.js	Assertion failure: script->code <= target && target < script->code + script->length, at jsopcode.cpp:5502
check-deltablue.js	Segmentation fault in `Writer::call()`, signaling failure to generate trace
check-earley-boyer.js	Assertion failure: (frameobj == NULL) == (*mTypeMap == JSVAL_TYPE_NULL), at jstracer.cpp:3168
check-raytrace.js	non-executable fragment
testReallyDeepNestedExit.js	Error: Assertion failed: got 197, expected 198

Table 5.3: Failed testcases and the reasons for failure

to code executed in the compiler-process. For this purpose we use the program instrumentation tool PIN [24]. The code coverage evaluation is based on runtime data produced by the automated 'jit-test' testsuite packaged with TraceMonkey.

Test Setup.

PIN is a tool which allows the user to register callback-functions that are called whenever a certain event, a routine-call, the execution of an instruction or a fork occurs. The calls to these user-supplied functions are dynamically inserted as the binary-image is loaded. This causes, depending on the inserted code, a significant performance-loss. On the other hand, it gives the user the ability to observe the execution of a program from within. It also offers an opportunity for dynamic modification of the analyzed code.

Process	Function count	Instruction count
Compiler	935k	251,106k
Executor	5k	150k

Table 5.4: Number of functions and instructions (executed during the first run of each function) after 1087 test cases.

We chose PIN for coverage-analysis since more traditional tools, such as Valgrind [22] or gcov, the coverage-tool shipped with gcc, were unsuitable for various reasons. GCov is not usable in our case since its output at runtime is written into one file per source-file regardless of the process that executes an instruction. Valgrind, which distinguishes between different processes, failed to correctly emulate the way in which we use the `clone()` system-call. Modifying these tools so that they yield useful results when operating on he modified version of TraceMonkey was considered to be out of scope.

To use PIN, callback-functions have to be defined in a shared object-file called a 'PIN-tool'. For our purposes we used a slightly modified version of the 'proccount' tool. Whenever a new routine is encountered while loading the program, proccount will put it on an internal list. Also, a callback-function is inserted that is triggered whenever the newly discovered routine executes. That way, the tool can keep track of invocations of the routine. In addition to the callback executed at each routine-call, the PIN-tool inserts a callback-function for each instruction found in each routine. This allows it to keep an instruction-counter for an individual function.

We modified this PIN-tool by removing the list of routines found while loading the image. This list would inevitably contain all functions used by the executor and the compiler, even when PIN was attached to only one process. Instead, we write the name of a routine and the number of instructions executed after one call to a file. While this is far from a suitable measurement of instruction-coverage, much less of any sort of path-coverage typically desirable for testing, it does give a rough approximation of the complexity of each process.

This measurement does, however, give an exact Listing of the functions executed by each process. Since there is no case where the control-flow within a function decides on whether a certain piece of code is executed by the executor or the compiler [1], this is enough to determine how much native code can potentially be run with execute-rights to JIT-code.

Test Results.

The results of the coverage evaluation are depicted in Table 5.4. They clearly show that the separation between executor and compiler reduces the amount of code executed with execute-permissions on any chunk of heap-memory significantly.

In addition, we picked a subset of twelve test cases of varying complexity (including Deep-Bails and nested traces), chosen empirically to cover various classes of bugs found during development. They produced a trace of 44 executed functions with a total of 1522 instructions for the executor. In the same set of test cases, the compiler called 1603 unique functions with a total of 3,572,795 instructions during their first execution. When running the complete set of

[1] With the exception of `switchContextToWait()`

regression-tests of 1087 test-cases, the number of functions executed by the executor rises to 5023 with 150,229 instructions. At the same time, the compiler-process has executed 935,337 functions with over 250 million instructions. Listing 5.2 shows the functions called by the executor in the course of the execution of ten of the above-mentioned twelve representative test-cases. The remaining two cases were not executed in this test-run since, because they caused either PIN itself or the running PIN-tool to experience instability. Since PIN shares its virtual memory with the process that is being observed and influenced, the presence of PIN alone may negatively influence the stability of the system. In particular, the memory-allocations performed by the remaining test-cases seemed to cause stability-issues. We suspect that this is due to the fact that PIN does not expect its stack to reside on the heap and thus potentially be subject to functions that modify the heap.

Listing 5.2: Partial listing of functions executed by the Executor-Process

```
1 function: __i686.get_pc_thunk.bx
2 function: _Z22VMPI_setPageProtectionPvjbb
3 function: JS_SetThreadStackLimit
4 function: _ZN2js18SetNativeStackBaseEPvi
5 function: _ZN2js19switchContextToWaitEPNS_6sharedEb
6 function: _ZN2jsL28SetPropertyByIndex_INTERCEPTEP9JSContextP8JSObjecti
7 PNS_5ValueEi
8 function: _ZL5PrintP9JSContextjP12jsval_layout
9 function: _Z32js_NewArgumentsOnTrace_INTERCEPTP9JSContextP8JSObjectjS2_
10 function: _Z22js_NewArgumentsOnTraceP9JSContextP8JSObjectjS2_
11 function: _ZL11array_sliceP9JSContextjPN2js5ValueE
12 function: _Z27js_NewNullClosure_INTERCEPTP9JSContextP8JSObjectS2_S2_
13 function: _Z17js_NewNullClosureP9JSContextP8JSObjectS2_S2_
14 function: _Z19js_obj_defineSetterP9JSContextjPN2js5ValueE
15 function: _ZN2js22IteratorMore_INTERCEPTEP9JSContextP8JSObjectPNS_5ValueE
16 function: _ZN2jsL12IteratorMoreEP9JSContextP8JSObjectPNS_5ValueE
17 function: _Z30js_CloneRegExpObject_INTERCEPTP9JSContextP8JSObjectS2_
18 function: _Z20js_CloneRegExpObjectP9JSContextP8JSObjectS2_
19 function: _Z14js_regexp_execP9JSContextjPN2js5ValueE
20 function: _Z27js_NumberToString_INTERCEPTP9JSContextd
21 function: _ZL25js_NumberToStringWithBaseP9JSContextdi
22 function: _Z26js_ConcatStrings_INTERCEPTP9JSContextP8JSStringS2_
23 function: _ZN2jsL27SetPropertyByName_INTERCEPTEP9JSContextP8JSObjectPP8
24 JSStringPNS_5ValueEi
```

While the call-trace of the compiler shows all functions related to JIT-compilation as well as the builtin-functions executed during a JIT-trace, the executor only shows functions that are necessary for the context-switch and the functions initially called when builtin-functionality is to be executed on trace. The trace may also include functions for memory-allocation such as `mmap()`. Most of the invocations in the executor are related to tasks such as string-concatenation, the setting and getting of object-properties or the execution of regular expressions. They remain on the

calltrace since they have been replaced with stubs that trigger a context-switch if they are called from the executor. Since these stubs are all generated by a small set of C++ macros, varying only in the number of parameters that are passed, this code is easy to audit. The rest of the executed code is either located in LibC or in the function `ExecuteTrace()`. This function encapsulates the start of the trace and the setting of execute-rights to newly compiled code-fragments and currently measures 63 lines of code.

5.3 Analysis of individual testcases

Finally, we will analyze the performance and behaviour of several test-cases in greater detail. We will discuss the causes of notable slowdowns compared to the baseline. We will also discuss test-cases that show very little performance-overhead and we will attempt to determine the roots of differing runtime-behaviour in the various test-cases.

InnerLoopIntOuterDouble

Listing 5.3: InnerLoopIntOuterDouble

```
1  function innerLoopIntOuterDouble() {
2      var n = 10000, i=0, j=0, count=0, limit=0;
3      for (i = 1; i <= n; ++i) {
4          limit = i * 1;
5          for (j = 0; j < limit; ++j) {
6              ++count;
7          }
8      }
9      return "" + count;
10 }
11 assertEq(innerLoopIntOuterDouble(), "50005000");
```

The testcase shown in Listing 5.3 is called innerLoopIntOuterDouble.js. The term 'OuterDouble' presumably refers to the internal type of the counter variable of the outer loop, which cannot be unified to integer due to the multiplication before the start of the inner loop. Apparently, this operation causes the engine to believe that it cannot safely use integer without risking an overflow.

The behavior of the script showcases a very favorable case for Just-In-Time compilation. The function has a computational complexity of approximately $1/2 * n^2$, as the expected result of the variable `count` (asserted by the javascript-function `assertEq()` in line 11 shows). As expected, the engine calls its compiler twice. Once to compile the outer loop and then again, after having executed the code of the inner loop several times in a rows, to compile the inner loop into a seperate fragment. Given that the upper bound of the inner loop is equal to the counter of the outer loop, the modified engine switches context from compiler to executor and back again relatively frequently in the beginning. After the inner loop has been compiled and linked into the outer loop (after about eight iterations of the outer loop), the behavior changes completely.

Beyond that point, both loops are executed natively in the executor, without a context-switch after the inner loop completes. This one, long trace-run causes the script to be much faster when JIT-compiled compared to an execution without JIT-compilation. Without JIT-compiler, the script took approximately 11867 milliseconds to complete. In a JIT-enabled test-run, the testcase containing the function passed in approximately 144 milliseconds.

Since there are no calls to builtin-functions, memory-allocations or other features that could cause a contextswitch to be performed after the inner loop is compiled, the control-flow of the modified engine does not deviate from the control-flow of the unmodified engine. The modified version is therefore not significantly slower than than the baseline. In fact, the modified version executed faster than the baseline in one test-run (144.372 compared to 144.535 milliseconds). Testcases like this one indicate that the architecture is fundamentally viable and that the existance of two interoperating processes alone does not cause punitive overhead.

TestAtomize

Listing 5.4 shows the function `TestAtomize()`. This testcase is, compared to the previous example, somewhat more complex. While it is computationally simpler, consisting of two loops that are consecutive instead of nested, the function `fromCharCode()` adds some complexity. It converts the number supplied as an argument into a unicode character. Being a builtin-function, it is implemented in native C++ and has to be executed by the compiler-process. Thus, it may cause the modified JIT-engine to switch context, even after the loop it is contained in has been compiled. Additionally, the array-access for which the string returned by `fromCharCode()` serves as index is implemented as yet another builtin-function with its own context-switch (`setPropertyByName()`). In the course of `setPropertyByName()`, the functions `js_AtomizeString()` and `Atomize()` are eventually called, giving the testcase its name.

The testcase is notably slower in the modified version than it is in the unmodified version. Using the unmodified version, the testcase takes 140.225 milliseconds to pass. When run by the modified engine, it takes 2903.687 milliseconds. The unmodified engine without JIT-compilation runs the testcase in 230.826 milliseconds. The performance-overhead observed here appears to be rather severe. Indeed, this test is part of a class of testcases with a runtime of over ten times that of the unmodified version. Observation has shown that memory-operations are performed during the execution of all members of this class of testcases. In the following, we will attempt to find possible causes of this runtime-overhead.

Listing 5.4: TestAtomize

```
1 function testAtomize() {
2     x = {};
3     for (var i = 0; i < 65536; ++i)
4         x[String.fromCharCode(i)] = 1;
5     var z = 0;
6     for (var p in x)
7         ++z;
8     return z;
```

```
9 }
10 assertEq(testAtomize(), 65536)
```

One suspect is, of course, the number of context-switches. The testcase `TestAtomize` executes at least three builtin-functions: `setPropertyByName()` and `String.fromCharCode()`. The iterator used in the second loop is also implemented as a native builtin-function. At least two builtin-functions are executed in the first loop. Of these, `setPropertyByName()` causes hashtable-lookups and insertions. The insertions into the hashtable may cause the mapping of memory. These memory-operations imply another context-switch. In total, that makes at least three context-switches in the worst-case during one iteration of the first loop and one context-switch during an iteration of the second loop. However, it seems hard to believe that three context-switches per iteration, each of which is implemented by a few hundred instructions, could cause such massive overhead.

Further analysis showed that, while the most time-consuming operations are near memory-allocations, they are not the allocations themselves. Using a debugger (GDB), the function `Chunk::init()`, which is responsible for managing the usage of memory-chunks, was identified as behaving abnormally in the modified version. Apparently, the engine fails to find memory-chunks in `Chunk::init()` that have been previously allocated and eventually allocates a new chunk to hold a value. This causes far more memory to be allocated than necessary. Not only does this cause a waste of memory-space, this problem also slows down the engine's memory-management. In `Chunk::init()`, a list containing hundreds of lost memory-'arenas' has to be iterated on whenever a hash-lookup takes place. This appears to be the main source of the slowdown observed in this testcase. Unfortunately, it is not clear when and why these memory-segments are allocated, since this phenomenon is not limited to testcases where arrays, hashes, or other objects are frequently accessed and written to. Rather, it seems to be related to calls to builtin-functions.

Check-crypto-md5

At first glance, this is a fairly complex testcase that implements the calculation of an md5-checksum of a string. As such, this can be seen as a 'real-world' application of the javascript-engine. Closer examination of the testcase shows that the testcase consists primarily of arithmetical and bit-shifting operations. There are also several loops containing calls to `charAt()`, a builtin-function that operates on a string and returns the character at a given position. This means that there is a significant number of context-switches in the course of the testcase's execution, depending on the number of characters within the given string. However, the testcase does not generate the vast number of empty arenas that are observed in `TestAtomize`. The reason for this may be that `check-crypto-md5` does not need to allocate memory for a large number of strings. Either way, no large-scale memory-operations and subsequent iteration through lists of memory-segments is observed. Also, there is a long-running, uninterupted trace-execution before the end of the testcase.

As a result, the observed overhead is very moderate. An execution takes an average of 56 milliseconds in the modified version. In comparison, the baseline takes 50 milliseconds while a

JIT-disabled test-run takes 183 milliseconds.

Check-crypto-aes

Although it is also implements a cryptographic algorithm, this testcase is quite different from the former. While it also passes successfully, it takes 719.7 milliseconds to finish when running in the modified engine. The baseline needs 99.4 milliseconds while a testrun without JIT-compilation takes 310.9 milliseconds. While several context-switches occur in the previously discussed testcase, `check-crypto-aes` causes the engine to switch context more frequently still. In addition to the context-switching caused by the execution of traces, there are frequent calls to `NewDenseUnallocated Array()` on the trace, which cause context-switches as well. This is unsurprising as the testcase operates on a set of arrays. The execution continues unremarkably, until at one point, presumably due to increased memory-pressure, the function `Refill FinalizableFreeList()` is called in the course of a `NewDenseUnallocated Array()`-call. This call coincides with the allocation of new shared memory. Nearing the end of the execution, more and more calls to `NewDenseUnallocatedArray()` include calls to `Refill FinalizableFreeList()` and consequential memory-allocations. This is likely a similar phenomenon to the wasteful arena-allocations in `TestAtomize`. At least, the control-flow appears to be similar. Much memory is allocated that is never used again but that is iterated over whenever an object is accessed. It is likely that many unneccesary array-objects are allocated which are stored in a list (presumably a list of objects that are to be freed).

The net-result is a performance-overhead of more than seven times compared to the baseline and over two times compared to a non-JIT execution.

Check-access-nsieve

The testcase shown in Listing 5.5 is a prime-numbers sieve. Computationally, the complexity-class lies within $O(m * log(log(m))$. The actual computation performed on the trace is very simple. It requires only an array, which is accessed frequently, the addition of several counters and some boolean logic. The code will spend most of its time between the lines 7 and 12. The only non-obvious operation performed here is a call to `EnsureDenseArrayCapacity()`, which, as the name implies, ensures that an array-index is within the bounds of the array. Other than that, the testcase consists of long, uninterupted traces. It is curious to note that not every iteration of the inner loop causes the builtin-function to be called. Most likely, the function is only called for indices that have not been previously encountered.

As there are no operations that affect the allocation of memory except the generation of the arrays in the function `sieve()`, the severe runtime-overhead encountered in testcases that trigger the previously mentioned memory-handling error does not occur here. Instead, we observe an overhead of less than two times (21 versus 11 milliseconds) compared to the baseline and a speedup of over ten times compared to a run without JIT-compilation. The runtime-overhead compared to the baseline can be explained as the overhead caused by frequent context-switching. The overhead is higher than in the other testcases that do not exhibit faulty memory-

management. However, there is also more context-switching than in the other 'fast' testcases.

Listing 5.5: Check-access-nsieve

```
function nsieve(m, isPrime){
    var i, k, count;

    for (i=2; i<=m; i++) { isPrime[i] = true; }
    count = 0;

    for (i=2; i<=m; i++){
        if (isPrime[i]) {
            for (k=i+i; k<=m; k+=i) isPrime[k] = false;
            count++;
        }
    }
    return count;
}
var ret = 0;
function sieve() {
     for (var i = 1; i <= 3; i++ ) {
         var m = (1<<i)*10000;
         var flags = Array(m+1);
         ret += nsieve(m, flags);
     }
}
sieve();
assertEq(ret, 14302)
```

Check-bitops-nsieve-bits

Listing 5.6 shows the testcase `Check-bitops-nsieve-bits`. Here, we can clearly see the effects that small changes of an algorithm can have on the runtime-behavior of the testcase. The algorithm itself is very similar to that of the previous testcase. The main difference between the two is that the array used to determine if a number is prime or not is far longer in this implementation. Additionally, the function `array_push()` is used to store a result (prime or not prime). However, the observed runtime-behavior is very different. Instead of a slowdown by a factor of two compared to the baseline (21 ms versus 11 ms), we observed a slowdown of 169 times (8792 ms compared to 52 ms in the baseline and 1344 ms in a JIT-less run). This very severe slowdown is caused by a previously unencountered memory-management error in the custom heap management library. We use this library as a replacement for the `dmalloc`-family of functions to facilitate the management of shared memory. This library stores chunks of memory in a sorted heap. Apparently, the library has a bug that causes the `Splay()`-function of the heap to iterate over far more memory-chunks than necessary. This aberrant behavior is

triggered in the previously seen builtin-function `EnsureDenseArrayCapacity()` as well as in the function `array_push()`, which was not used in the previous testcase. This indicates that it is the increased size of the array, and not the use of `array_push()` that causes the problem.

We were unable to locate the root-cause of the bug, but manual analysis indicates that while unrelated, it is similar in nature to the previously mentioned memory-management issue that occurs in `NewDenseUnallocatedArray()` and which causes the allocation of superfluous segments of shared memory. While this bug occurs in the memory-management functions of the engine itself, the bug encountered occurs in a function added specifically as part of our modifications. As such, these bugs may hint at one basic problem in two different implementations.

Listing 5.6: Check-bitops-nsieve-bits

```
1 var result = [];
2
3 function primes(isPrime, n) {
4   var i, count = 0, m = 10000<<n, size = m+31>>5;
5
6   for (i=0; i<size; i++) isPrime[i] = 0xffffffff;
7
8   for (i=2; i<m; i++)
9     if (isPrime[i>>5] & 1<<(i&31)) {
10      for (var j=i+i; j<m; j+=i)
11        result.push(isPrime[j>>5] &= ~(1<<(j&31)));
12      count++;
13    }
14 }
15
16 function sieve() {
17     for (var i = 4; i <= 4; i++) {
18         var isPrime = new Array((10000<<i)+31>>5);
19         primes(isPrime, i);
20     }
21 }
22
23 sieve();
24
25 var ret = 0;
26 for (var i = 0; i < result.length; ++i)
27   ret += result[i];
28
29 assertEq(ret, -211235557404919)
```

TestDoubleComparison

The testcase `testDoubleComparison` is perhaps the simplest testcase so far. It consists of a loop that is executed 500000 times. The loop contains a `switch`-statement with a static test-condition of $1/0$. The only `case`-statement is the value `Infinity`. After completion of the loop, the string `'finished'` is returned. The return-value is checked in an assertion and an interal function of the engine is called to determine the number of side-exits taken, which has to be 1. As expected, the modified architecture has a relatively small overhead compared to the baseline (3.9 ms versus 3.6 ms and 181.7 ms when running without JIT-compilation). The loop is executed as a single, unremarkable trace that does not call any builtin-functions. The observable overhead (a factor of 1.083) is caused by the provisioning of shared memory, process-creation and similar administrative tasks which may influence the runtime of short testcases but become neglectible for longer-running traces.

Check-3d-morph

Listing 5.7 shows the testcase `Check-3d-morph`, a relatively complex testcase using mathematical functions. The runtime-behavior is similar to that observed in testcase 5.6. Again, `EnsureDenseArrayCapacity()` is called, which in turn causes a long search for a suitable block of memory in the custom heap management library. Additionally, the function `growSlots()`, responsible for increasing the size of an array, executes a very long-running loop in which the capacity of an array is increased incrementally by a large margin. The size of the array is increased by thousands of elements with each call of the function and each new array-slot equals to one loop-iteration. All in all, our modifications caused the performance to drop from 20.73 milliseconds (345 ms without JIT) to 4271 milliseconds.

Listing 5.7: Check-3d-morph

```
1 var loops = 15
2 var nx = 120
3 var nz = 120
4
5 function morph(a, f) {
6     var PI2nx = Math.PI * 8/nx
7     var sin = Math.sin
8     var f30 = -(50 * sin(f*Math.PI*2))
9
10    for (var i = 0; i < nz; ++i) {
11        for (var j = 0; j < nx; ++j) {
12            a[3*(i*nx+j)+1]     = sin((j-1) * PI2nx ) * -f30
13        }
14    }
15 }
16
```

```
17
18 var a = Array()
19 for (var i=0; i < nx*nz*3; ++i)
20     a[i] = 0
21
22 for (var i = 0; i < loops; ++i) {
23     morph(a, i/loops)
24 }
25
26 testOutput = 0;
27 for (var i = 0; i < nx; i++)
28     testOutput += a[3*(i*nx+i)+1];
29 a = null;
30
31 /* not based on any mathematical error calculation.*/
32 acceptableDelta = 4e-15
33
34 assertEq((testOutput - 6.394884621840902e-14) < acceptableDelta, true);
```

Summarizing, the largest runtime-impact occurs when the size of arrays is modified. While this decrease in performance is unfortunate, we have shown that it is related to an implementational flaw that is triggered when the capacity of arrays or objects that act like containers is manipulated. Finding the root-cause of this flaw is likely a significant engineering-challange, but the condition is not inherent to the architecture proposed here.

CHAPTER 6

Conclusion

6.1 Limitations

The primary limitation of the architecture of *Lobotomy* is that there is an inherent runtime overhead associated with switching between the two contexts. In the current implementation, the total overhead can reach 550 percent. We believe, however, that careful optimization and analysis of the security-requirements could help reduce the performance penalty. In particular, many builtin-functions may prove to be harmless. This is the case for functions that do not compile nested traces and that do not interact with data controlled by the user.

Further decrease in overhead may be archived by improving the handling of shared memory and by integrating it into the memory-management system of TraceMonkey. This is in particular the case since the current proof-of-concept implementation has some bugs related to memory-management that increase the runtime drastically. Without these bugs, it is likely that the runtime-overhead would level out at around a factor two at most, as shown by the testcases that do not trigger this error.

While some stability-issues currently exist, they could be removed by adding awareness of the separated execution-contexts to other parts of TraceMonkey than the Tracer itself. In the current implemenation, the logic that translates virtual-machine instructions to the intermediate language used by nanojit, and that eventually is compiled into the trace, is mostly unchanged. Adapting this logic with separated processes in mind would not only increase the stability of the architecture but also increase its performance.

Another potential issue is the relative age of Firefox 5. The javascript-engine was changed significantly in newer versions of TraceMonkey and its successors. However, the foundations layed down in this thesis are still applicable, since they can be applied to any system that dynamically compiles code.

6.2 Future Work and Conclusion

In this thesis we presented *Lobotomy*, an approach to hardening a JIT-engine's design against JIT-spraying attacks. Simply put, the main idea is to split compilation and execution of JITted code into two strictly separated parts to reduce the attack surface. We implemented this approach as a working proof-of-concept in TraceMonkey, the JIT-engine of Firefox. We proved that the fully separated architecture proposed here reduces the attack-surface of TraceMonkey for JIT-spraying by at least 100 times. We accomplish this without having to iterate over large lists of code-fragments, which keeps the runtime-overhead relatively low. Nonetheless, there are no code-fragments that are mapped with `RWX`-permissions, because the virtual memory of the two processes is fully separated.

Still, future work will focus on further optimizing our implementation performance-wise, with the ultimate goal of integrating it into a recent version of Firefox. It should be noted that adapting an existing architecture to new requirements is often a harder engineering-task than building a new architecture with these requirements in mind. It may be easier to create a new javascript-engine in which the idea and concepts presented in this thesis are already built in. The difficulty of that approach would be to optimize the engine to such a degree that it yields results in various performance-tests that are at least comparable to that of modern, widely used engines. Otherwise, a proof that the separation of execution and compilation of JIT-code has no drastic impact on performance by itself would be difficult to accomplish. For this reason, we chose to modify an existing architecture even though this has introduced some subtle bugs which must be addressed in a future version of the system.

Lobotomy is available as an open source project to give interested researchers and engineers the opportunity to advance our first prototype.

Bibliography

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] Alexa. Top Sites. `http://www.alexa.com`. last accessed: 20.02.2013.

[3] Piotr Bania. Jit spraying and mitigations. *arXiv preprint arXiv:1009.1038*, 2010.

[4] Michael Bebenita, Mason Chang, Karthik Manivannan, Gregor Wagner, Marcelo Cintra, Bernd Mathiske, Andreas Gal, Christian Wimmer, and Michael Franz. Trace based compilation in interpreter-less execution environments. Technical report, ICS-TR-10-01, University of California, Irvine, 2010.

[5] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying. *Blackhat, USA*, 2010.

[6] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document, 2009.

[7] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. Technical report, Technical report, Secure Systems Lab at Stony Brook University, 2007.

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[9] Ping Chen, Yi Fang, Bing Mao, and Li Xie. Jitdefender: A defense against jit spraying attacks. In *SEC'11*, pages 142–153, 2011.

[10] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, volume 81, pages 346–355, 1998.

[11] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. Jitsec: Just-in-time security for code injection. In *WISSEC 2010*, 2010.

[12] Solar Designer. Getting around non-executable stack (and fix). Bugtraq mailing list, http://seclists.org/bugtraq/1997/Aug/63 (last accessed 9th Oct. 2011), 1997.

[13] DVLabs. EUSecWest Pwn2Own Contest. `http://dvlabs.tippingpoint.com/blog/20120`, 2012.

[14] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, pages 88–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Hiroaki Etoh and Kunikazu Yoda. Propolice: Improved stacksmashing attack detection. *IPSJ SIG Notes*, 75:181–188, 2001.

[16] Mozilla Foundation. Js trace test suite. `http://dxr.mozilla.org/mozilla-central/source/js/src/jit-test/README`. last accessed: 21.09.2013.

[17] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[18] Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Agile Conference, 2006*, pages 6–pp. IEEE, 2006.

[19] Kingcope. Attacking the windows 7/8 address space randomization. Wordpress page, https://kingcope.wordpress.com/2013/01/24/attacking-the-windows-78-address-space-randomization/ (last accessed 26th Jul. 2013), 2013.

[20] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 29–44. IEEE, 2010.

[21] Mozilla. Firefox Bug 506693. `https://bugzilla.mozilla.org/show_bug.cgi?id=506693`, 2012.

[22] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[23] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):365, 1996.

[24] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, and Andrew Sun. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *In International Symposium on Microarchitecture*, pages 81–92. IEEE Computer Society, 2004.

[25] Mathias Payer. I control your code. In *Proceedings of the 27th Chaos Communication Congress (27c3)*, 2010.

[26] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Microsoft Tech Report MSR-TR-2008-176*, 2008.

[27] G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 60 –69, dec. 2009.

[28] Chris Rohlf and Yan Ivnitskiy. The security challenges of client-side just-in-time engines. *Security & Privacy, IEEE*, 10(2):84–86, 2012.

[29] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, volume 12, 2012.

[30] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[31] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[32] Alexey Sintsov. You can't stop us: latest trends on exploit techniques. In *CONFidence 2010, Krakow*, 2010.

[33] Richard Stallman, Roland H Pesch, Stan Shebs, et al. *Debugging with GDB*. Gnu Press, 2002.

[34] PaX Team. Pax address space layout randomization (aslr), 2003.

[35] Tao Wei, Tielei Wang, Lei Duan, and Jing Luo. Secure dynamic code generation against spraying. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 738–740. ACM, 2010.

[36] Yang Yu. Dep/aslr bypass without rop/jit. In *CanSecWest 2013*, 2013.