Karlsruher Institut für Technologie

Lightweight Introspection for Full System Simulations

Diplomarbeit
von

Jonas Julino

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dipl.-Inform. Marc Rittinghaus

Bearbeitungszeit: 2. September 2013 – 1. März 2014

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Jonas Julino
Karlsruhe
March 1, 2014

Deutsche Zusammenfassung

Betriebssysteme und Anwendungen können sehr komplex sein und die Wechselwirkungen zwischen ihnen in einem laufenden Computersystem erhöhen die Komplexität weiter. Vorhersagen über das Laufzeitverhalten solcher Anwendungen sind aus menschlicher Perspektive nahezu unmöglich. Um dennoch das Verhalten analysieren zu können, ist es naheliegend die Ausführung von Programmen zu beobachten, anstatt zu versuchen sie vorherzusagen. Einfache Eigenschaften wie die Laufzeit eines Programms lassen sich leicht messen. Um aber detailierte Informationen über den Zustand eines Betriebssystems oder dessen Programme zu erhalten, wie zum Beispiel die Werte von bestimmten Variablen, ist ein komplexerer Ansatz nötig.

Zur Überwachung eines Programms, ist es möglich dieses so zu modifizieren, dass es Werte von Variablen zur Laufzeit ausgibt, oder es kann mittels eines *Debuggers* beobachtet werden. In beiden Fällen führt dies zu einer Änderung an dem zu überwachenden Programm. Das Starten eines Debuggers hat Einfluss auf die vorhanden Ressourcen und die zeitlichen Abläufe in einem Betriebssystem. Für den Fall, dass Software Haltepunkte verwendet werden, ist es zusätzlich auch nötig den Speicher des Programms, das überwacht werden soll, zu verändern.

Bei der Verwendung eines Simulators, der ein komplettes Computersystem simuliert, ist es möglich auf jeden Teil dieses *Gastsystems* zuzugreifen, ohne dadurch das Laufzeitverhalten des Gastes zu beeinflussen.

Lediglich Zugriff auf die Daten eines Gastes zu haben, wie zum Beispiel auf den Inhalt seines Speichers, ist aber nicht ausreichend, um einen tieferen Einblick in den Zustand der ausgeführten Programme zu erhalten. Das Interpretieren dieser Daten ist aufgrund der *Semantischen Lücke* zwischen dem Simulator, der Zugriff auf den Speicher und die Register des Gastes hat, und einem Programm, das im Gast ausgeführt wird und seinen Zustand nach eigenem Ermessen im Arbeitsspeicher und den Registers enkodiert, nicht einfach.

Introspection Werkzeuge, wie *Volatility* [55], schließen die Semantische Lücke und ermöglichen es ein Abbild eines Gastsystemes zu analysieren, um zum Beispiel alle momentan laufenden Prozesse auflisten zu können.

Das Analysieren von vollständigen Abbildern ist insbesondere für forensische Analysen nützlich. Um ein laufendes System kontinuierlich überwachen zu können, ist es allerdings nötig solche Analysen regelmäßig durchzuführen.
Damit dabei keine Ereignisse von kurzer Dauer verpasst werden, muss nach jeder Veränderung des Gastzustandes, der zu einem veränderten Introspection-Ergebnis führen kann, die Analyse erneut ausgeführt werden. Im Bezug auf Ansätze, die das Speicherabbild als Grundlage verwenden, ist jeder Schreibzugriff auf den Speicher ein solches Ereignis. Aus diesem Grund sind solche Verfahren zu langsam, um in einem solchen Szenario verwendet zu werden.

Das Ziel dieser Arbeit ist es einfach zu benutzende Werkzeuge zur Verfügung zu stellen, die es ermöglichen ein laufendes Gastsystem und seine Anwendungen zu beobachten. Die Simulation soll dabei möglichst wenig verlangsamt und die Beeinflussung des Gastes vermieden werden.

Um dies zu erreichen, entwickeln wir einen Ansatz für event-basierende Introspection, der sich Debugsymbolen bedient, um die Semantische Lücke zu schließen. Er bietet die Möglichkeit Haltepunkte im Gastbetriebssystem und dessen Anwendungen zu setzen, wie es von Debuggern bekannt ist. Anstatt den Zustand des Gastes nach jedem Schreibzugriff zu untersuchen, wird dies nur an den Haltepunkten getan, was die Anzahl an nötigen Analysen stark reduziert.
Außerdem ist es nicht nötig den vollständigen Zustand des Gastes an jedem Haltepunkt zu analysieren. Da Haltepunkte jeweils ein Ereignis im Gast, wie zum Beispiel die Erzeugung eines neuen Prozesses, repräsentieren, ist lediglich der Zugriff auf Variablen und Strukturen nötig, die sich durch dieses Ereignis verändert haben könnten.

Der Beitrag dieser Arbeit ist der Entwurf und die Implementierung eines event-basierenden Introspection-Ansatzes, der es ermöglicht Ereignisse im Gastbetriebssystem und dessen Anwendungen zu beobachten. Die Implementierung kann für verschiedene Plattformen und Formate erweitert werden. In der vorliegenden Arbeit wird konkret allerdings nur die Implementierung für ein AMD64 basierendes Linux Gastsystem beschrieben, das mittels *Quick Emulator* simuliert wird. Durch die Anwendung unseres Introspection-Ansatzes wurde der verwendete Simulator bei der Beobachtung aller Wechsel zwischen den Gastprozessen sowie deren Erstellung und Löschung in unseren Tests um weniger als 10% verlangsamt.
Diese Arbeit beschäftigt sich auch mit den Einschränkungen von debugsymbol-basierenden Introspection-Ansätzen, wie zum Beispiel Probleme bei der Verwendung von optimierten Programmen, und zeigt Lösungswege für diese auf.

Abstract

Full System Simulators can be used as convenient tools for tracking and measuring runtime behavior of complete computer systems. One of their advantages is the ability to observe the complete state of the simulated system without modifying it. Unfortunately it is not sufficient to be able to measure, but it is essential to interpret the results. The fundamental problem of introspecting the state of the simulated system is the semantic gap; understanding the high-level meaning of a register value or the memory content is difficult from the perspective of the simulator. Current bridges for this gap have different drawbacks: (i) they are only intended to be used with snapshots of a system, (ii) they require modification of the simulated system or (iii) they are extremely slow.
This thesis proposes a new event-based introspection approach which can be used to observe the high-level state of a simulated system continuously, without requiring modifications to it. The resulting slowdown is moderate compared to other approaches.

Contents

Chapter 1
Introduction

Operating Systems and programs can be very complex on their own and the interactions between them in a running system increase this complexity further. Predicting their exact runtime behavior is almost impossible — at least from a human being's perspective. A solution for this is to observe the behavior during runtime instead of trying to predict it. While simple characteristics such as the time, necessary to execute a program, can be measured easily, detailed information on the state of an Operating System (OS) or program, such as values of certain variables, requires a more complex approach.

Monitoring software by modifying it to output values of certain variables or by using a *debugger* results in modifications to the target of observation. Starting a debugger alters the available resources and the timing in the OS. In case of using software breakpoints, the code of the debugged program is modified during runtime.
With a full system simulator it is possible to access every part of a simulated system (registers, memory, ...) at any point in time without modifying its runtime behavior. Unfortunately, in order to determine the value of a variable within a simulated program, it is not sufficient to be able to access its simulated resources, but the semantic meaning of the data seen is required as well.
Acquiring this meaning is not easy due to the *semantic gap* between the simulator which simulates a *guest system* on the hardware level and a program running inside the guest which encodes its internal state in memory and registers at its own discretion. *Introspection* approaches which bridge the semantic gap such as *Volatility* [55] allow to analyze a snapshot of a guest system and to extract high-level information on the state of the (simulated) OS and programs. Such snapshot based introspection approaches can be particularly useful for post-mortem analysis of computer systems. However, in order to analyze a running system continuously, it is necessary to execute them repeatedly.
Every change of the guest state, possibly resulting in a different introspection result, requires to re-execute these approaches in order to avoid missing events of short duration. For approaches accessing the memory image of a guest, each write instruction is such a change. Hence these solutions are much to slow to be used in this scenario.

The objective of this thesis is to provide a tool chain easily usable for introspecting a running guest system including its applications. The performance impact on the simulation speed is to be minimized as far as possible and the approach should be able to completely avoid affecting the guest system.

To achieve this goal, our work introduces the idea of event-based introspection utilizing debugging symbols. This enables the setting of breakpoints in the guest OS and its applications as known in debuggers. By introspecting the guest system only at breakpoints, the number of introspection runs can be reduced extremely as it is no longer necessary to analyze the guest system completely after each write instruction of the guest.
Additionally there is no need to retrieve the complete guest state at each breakpoint. The breakpoints represent an event in the guest such as the creation of a new process, hence it is only necessary to access variables or structures which might be changed due to this event.

The contribution of this work is the design and implementation of a fast event-based introspection approach capable of observing events in a guest OS and its applications. While the implementation is extensible to different platforms and formats, this thesis targets only the implementation for an AMD64 based Linux guest which is simulated by Quick Emulator [5]. Applying our approach, the runtime of the simulation is increased by less than 10% when introspecting a Linux kernel to record task switch, create and delete events in our benchmarks.
This work also discusses the limitations of debugging symbol based introspection approaches — such as difficulties arising with optimized executables — and presents ways to cope with them.

The following document is organized as follows: Chapter 2 covers background information and related work on full system simulators, current introspection approaches and debugging symbols. Subsequently, Chapter 3 analyzes current methods for event-based extraction of high-level information from a guest system and discusses requirements to avoid guest modifications. The resulting insights are used to design an introspection approach in Chapter 4 and to describe an implementation in Chapter 5. Chapter 6 contains a performance benchmark of the implementation and describes simple examples for introspection done by using the tool chain. Additionally, the limitations of the approach are discussed and possible solutions are offered. This thesis closes with a conclusion and an overview on future work in Chapter 7.

Chapter 2
Background and Related Work

Full system simulators are useful tools for development and measurement tasks for not yet existing hardware or hardware which does not provide the required interfaces to access certain information. Because of the increased computation power and improved simulation techniques, it is meanwhile possible to interactively use quite complex computer systems simulated within them. We discuss the different design approaches and their existing implementations in Section 2.1.
While full system simulators can provide detailed information on the hardware state, the high-level semantic information on running computer systems, e.g., which processes are currently executed, is also of interest — especially for forensic analysis. Most approaches for forensic investigations such as FATKit [30] were initially intended to work with real world systems, but they can be used for simulated systems, too. In the context of full system simulation, this analysis is called *introspection* and we cover this topic in Section 2.2.
When trying to understand the state of a computer system it is necessary to obtain knowledge on the structures which represent information within this system. *Debugging symbols* contain such information and can hence be helpful for introspection. Section 2.3 contains some insights in the structure of widespread debugging file formats.
The last part of this chapter consists of Section 2.4 describing programs and libraries we used for this thesis and Section 2.5 containing terms and definitions we refer to in further chapters.

2.1 Full System Simulators

Full system simulators are tools to simulate a complete computer system including its devices. The granularity of the simulation can reach from *micro-architectural*, *cycle-accurate* and *deterministic* to only *functional correct* (instruction set level) simulations [8, 43, 61], depending on the objectives and settings of the individual simulator. While some authors refer to these only functional correct simulations as *emulation*, we use the term *simulation* independent from the level of detail in this document.

The range of applications for full system simulation includes development for platforms not yet existing in hardware, debugging and development of OSes, running legacy software on computers with a different Instruction Set Architecture (ISA) and instrumentation of virtual computer systems [32, 38].
If the aim is only the execution of a single program, a few full system simulators also allow system-call emulation [7], i.e., they can simulate the behavior of an OS interface instead of a complete system. It is unnecessary to simulate devices and the complete OS, as guest system-calls can often be mapped to host system-calls.
The first part of this section covers the basic techniques used to implement full system simulators (§ 2.1.1). The second consists of a short overview on common full system simulators and their features (§ 2.1.2). This section ends with a detailed discussion of the simulator used in this work (§ 2.1.3).

2.1.1 Full System Simulation Basics

Full system simulators aim to simulate a guest computer system on a host computer. The ISAs of the guest and the host may differ, but even if they are equal, it is impossible to execute the unmodified guest code on the host. If we execute guest instructions directly on the host, they would, e.g., when issuing a software interrupt, influence the host CPU instead of the virtual guest CPU. Furthermore the host CPU is not aware of the guest's virtual address translation structures and settings, hence all the guest address translation must be simulated by additional host code.
To overcome this issue there are two widely used modes of operation for a full system simulator: *Interpretive Execution*, which is typically used for more complex and detailed simulations, and *Dynamic Binary Translation*, which is most frequently used to achieve a high simulation speed.
While the slowdown of Dynamic Binary Translation based simulations can be below factor 10 compared to the runtime on physical hardware [47, 59], interpretive simulated systems are typically three or four orders of magnitude slower than real hardware [47]. However, there are interpretive simulators such as *Bochs* [32] which focus on speed instead of detailed simulations. Bochs is "only" 30 times slower than the Dynamic Binary Translation based Quick Emulator [5].

Interpretive Execution

When using interpretive execution, the state of the simulated guest is located in the host's main memory, in particular the CPU registers are represented by a location in the host's Random Access Memory (RAM).
The simulator code fetches and decodes each guest instruction one by one and modifies the guest's system state, e.g., the value of the Instruction Pointer (IP), like executing the individual instruction on a non-simulated system would. This whole process can be implemented as a huge code loop [32], executed for each instruction to interpret. For micro-architectural and cycle-accurate simulations, e.g., when simulating processor pipelines, the influence of one guest instruction can overlap with the following instructions and a detailed state machine is required for the representation of the guest CPU.

When the modeling of the instructions is done by writing a high-level language, like it is done in Bochs, the simulator can be compiled for every host ISA supported by the compiler used, without additional effort [32]. In addition, it is easy to instrument such simulators as the program flow within the simulator is rather simple and the code necessary for the instrumentation can be written completely in a high-level language.

Dynamic Binary Translation

Instead of interpreting each guest instruction consecutively, they can be translated to code directly runnable on the host CPU. Translating binary instructions to other binary instructions is called Binary Translation. This can be used to translate between different ISAs or to insert additional code into the instruction stream, as done in instrumentation tools like *Valgrind* [42] or *Pin* [36]. A further range of use is the support for legacy software after an ISA switch [1].
The translation of the binary instructions can either be done off-line (*Static Binary Translation*) or during runtime (*Dynamic Binary Translation*). Tools like ATOM [52] which use Static Binary Translation (SBT) translate a program's instructions and create a new executable file consisting of these translated instructions. The newly created binary is then executed. *Pure* SBT is difficult[1] and in general even impossible, as, e.g., supporting self-modifying code is impossible [1].
Because of the limitations of SBT and the difficulty to identify all code which will be executed during the runtime of a complete computer system [59] — programs might even be loaded during runtime from the internet — SBT is not typically used in full system simulation approaches.

When using Dynamic Binary Translation (DBT) the translation is done on a Basic Block-level. A Basic Block (BB) contains a number of machine instructions which must be executed sequentially, hence it may start on an arbitrary instruction, but it must end on an instruction possibly modifying the control flow, e.g., a jump [11]. Because of the sequential execution flow, it is easy to identify the instructions within a BB once the start address is known.
If the simulator must execute a guest instruction, a complete BB is translated into host code and executed directly on the host CPU afterwards. The end of each translated BB contains instructions to return to the simulator's code. Once the execution flow has returned to the simulator, it selects a new BB according to the current instruction pointer and the process starts over again. In case an exception or interrupt occurs while executing a BB, the simulator passes this event to the simulated CPU. The simulator then modifies the CPU state and in particular the IP according to the ISA's specification, e.g., by using an interrupt vector table. Subsequent to this, the normal execution can continue and the simulator selects a BB for the possibly modified IP. See Figure 2.1 for a graphical representation of the described steps.

[1] Common, von Neumann based, computer architectures store their data and machine instructions in the same memory. Distinguishing between instructions and data is not possible in general [26], there are however approaches which try to solve this for some practical scenarios where the machine code is

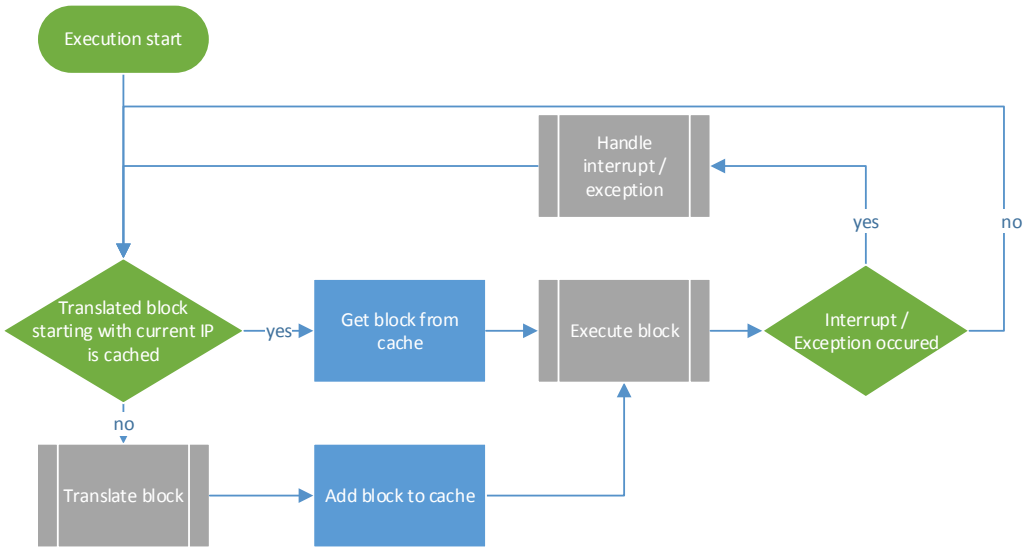

Figure 2.1: The fundamental execution flow in the dynamic binary translation process in a full system simulator. To execute guest instructions a complete BB, starting at the current IP, is translated and executed. The translation overhead is reduced by memorizing already translated BBs.

Assuming a BB has already been translated, the underlying guest instruction did not change and the already translated BB was stored somewhere, there is no need to translate it again [11]. To avoid executing instructions which are out-of-date, e.g., because of self-modifying code, the simulator must invalidate cached BBs when the underlying guest instructions change [59]. Although the translation of BBs causes additional work, DBT can be fast compared to interpretive execution, because of the reusing of cached BBs.

When translating guest to host instructions and executing them afterwards, there are some pitfalls an implementation of a full system simulator must avoid. The following paragraphs cover the problems arising in the translation and the execution of BBs.

Translation In case of full system simulation the actual translation consists of two different tasks:

1. Translating from the guest ISA to a possibly different host ISA.
2. Exchanging certain guest instructions which modify the systems state with host instructions which modify the *virtual* system state [59].

While the translation between ISAs is not always necessary, the modification of instructions must always be executed. The following categories of instructions to modify can be identified:

subject to some restrictions such as not to include dynamically modified or created code [1, 26].

Instructions Accessing Memory If an instruction accesses guest memory, the address used is in fact an offset into the guest memory which is located at an arbitrary address in the host's RAM. Therefore the guest address cannot be used directly within a host instruction, but instead the address must be translated to a valid host address before it can be used. Assuming the guest memory is represented by a contiguous memory region, this translation can be done by adding the start address of the region to the guest address.

However, a lot of processors contain a Memory Management Unit (MMU) which allows the use of virtual address spaces. As the translation between virtual and physical addresses on real hardware is done by the CPU, it is necessary to add code in front of each translated memory accessing guest instruction which performs the translation from guest virtual to guest physical addresses through a simulated MMU [59]. After adding the offset to translate the guest physical to a host virtual address, the addresses can be interpreted by the host MMU and hence they can now be used within the host code to access guest memory. Figure 2.2 summarizes the steps necessary to translate a guest virtual address to a host physical address.

As a lot of instructions access memory and the address translation is embedded in their executions, this causes a significant slowdown of the execution, even when using a well designed translation buffer. Witchel and Rosenblum [59] claim that their full system simulator *Embra* is slowed down by the factor 3 to 4 due to address translation.

Figure 2.2: The simulator translates guest virtual to guest physical addresses by simulating a MMU in software. The subsequent translation (e.g., adding an offset) redirects the access to a guest physical address to an address within an area of the simulator's virtual address space which contains the guest's physical RAM content. The host's MMU can translate the resulting host virtual addresses to host physical ones, just like it does it for every other virtual address the simulator's code accesses.

Privileged Instructions Translating privileged instructions to privileged instructions is not typically recommended, as a full system simulator does not normally run in a privileged mode and even if so, the resulting effect would not be correct with respect to the simulation result.

Instead these instructions must be translated into code modifying the simulated CPU's state. Changing the page directory on a simulated X86 ISA for instance must be translated to instructions notifying the *virtual* MMU, not the one from the host.

Software Interrupts As for privileged instructions, the simulator needs to translate software interrupts to host instructions modifying the state of the guest's CPU. The

generated translation must contain (i) a possibly necessary switch of the CPU's mode, (ii) the saving of certain registers to the stack and (iii) the loading of an address of an interrupt handler into the IP register.

Execution After a BB was translated it can be directly executed on the host and all translated guest instructions result in modifications of the guest's state. However, as the translated BB represents the behavior of the guest code on real hardware, executing them can result in exceptions, e.g., because of a devision by zero. In context of full system simulation we also need to deliver interrupts to the guest system, i.e., we must interrupt the execution of the guest and continue its execution in another context. Both cases require special handling within the simulator:

Exceptions The simulator must ensure that exceptions which occur during the execution of a translated BB are forwarded to the virtual guest CPU and not to the host CPU as the exception was caused logically by the guest code.

During the execution of a translated BB it is not necessary to keep all context and registers up-to-date, e.g., the IP. As long as the IP is not used inside the BB, it is sufficient to update its value at the end of the BB. While the lazy updating of registers yields less host instructions and hence a faster execution, it causes a problem in case of an exception.

As an exception may occur anywhere in a BB the simulator must be able to update the current virtual CPU state, if lazy updating is used [5].

Interrupts Devices issue interrupts in order to interrupt the execution of the CPU. On real hardware such interrupts can occur between two arbitrary instructions. To simulate interrupts of virtual devices correctly, the simulator must add a check for pending interrupts before each translated guest instruction.

However, to reduce the performance penalty, this check for pending interrupts can be postponed to the end of the BB. This reduces the amount of necessary checks to one per BB and additionally eases the handling of these interrupts as the whole guest CPU state is guaranteed to be up-to-date at the start and end of a BB. The disadvantage is that the interrupts can now be delayed slightly and hence this solution does not represent the exact timings within a real world system; but for a only functional correct simulation, that is not an issue.

2.1.2 Overview of Existing Full System Simulators

Dynamic Binary Translation is used in several different full system simulators like for example *simOS* [47] with the *Embra* [59] extension or *SimNow* [2]. *Bochs* [32] is an example for interpretive execution. The following sections give an overview of various popular full system simulators. We omit *Quick Emulator* [5] in this section as we discuss it in more detail in Section 2.1.3.

Simics

Simics [37, 38] is a deterministic full system simulator developed by Wind River. It comes with support for functional simulation of many well known ISAs, including X86, ARM and PowerPC. To allow cycle-accurate simulations Simics is able to connect to hardware models written in a hardware description language.

An interesting feature of Simics (called *Simics Central*) is to synchronize several simulated systems and to span a virtual network between them. This allows to simulate complete networks of computers without the problems caused by timing issues, between simulated components (at different simulation speeds) and real world components, e.g., switches and routers [38]. An example for such an evaded problem is a TCP timeout which could occur between a simulated system and a real-world system, as the wall-clock time needed by the simulated system to answer to a packet may exceed the timeout.

A disadvantage of Simics is that its source code is not available to the public, hence extending Simics is restricted to using its public interfaces. However, these interfaces allow influencing the simulator on a lot of different events, e.g., on every executed instruction.

MARSSx86

MARSSx86 [43, 45] is a cycle-accurate full system simulator only available for the X86 and AMD64 ISA. It supports in-order and out-of-order multi-core processor simulations in connection with a detailed simulation of coherent caches [43].

It was developed based on the source of Quick Emulator and its detailed CPU model originates from PTLsim [61]. To speed up the execution of runtime phases, not of interest for the user, MARSSx86 can switch between its own cycle-accurate simulation and Quick Emulator (QEMU)'s faster but only functional correct mode during runtime [43, 44].

Gem5 Simulator

Gem5 Simulator [7] offers deterministic full system simulation on different levels of detail for a variety of platforms including ARM and X86. CPU and memory models can be selected depending on the requirements regarding speed and precision. The simulation framework it adopted from M5 [6] and the memory simulation system from GEMS [39].

This simulator represents ISAs by a group of C++ classes (one for each instruction of the ISA) derived from a base class and a function which disassembles a machine instruction and returns an instance of the corresponding class. These instances can be used by calling their associated functions, e.g., `execute()`.

Although the simulation speed of gem5 Simulator can be increased by using the simplest available models, this simulator is "still considerably slower" [7] than DBT based approaches like QEMU or SimNow.

Like Simics and M5, gem5 Simulator supports simulation of several computers and a network connecting them. The simulation stays deterministic when using this feature [7].

2.1.3 Quick Emulator

Quick Emulator (QEMU) [5] is an instruction set level full system simulator with focus on speed and portability. The supported CPU architectures include (but are not limited to) X86, ARM and PowerPC in arbitrary combination as host and guest ISAs. To allow debugging of an OS running inside the simulator, QEMU includes a server stub for GNU Debugger (GDB).
QEMU uses DBT whose basic approach we already described in Section 2.1.1. Within QEMU the term *Translation Block (TB)* is used to refer to a Basic Block. We stick to this nomenclature. The DBT approach of QEMU adds some extensions to the basic approach, e.g., *block chaining* known from Shade [11] or *helper functions* used in Embra [59].
The following paragraphs describe the important extensions and differences during the translation and the execution of TBs. Afterwards we discuss the accuracy and determinism of QEMU.

Generation of Translated Blocks: The Tiny Code Generator

Tiny Code Generator (TCG) is the component of QEMU which is responsible for translating TBs of guest instructions into host instructions. The important differences of QEMU's approach are the use of an *intermediate format*, the support for *TCG helper functions* and the introduction of *static CPU state*.

Intermediate Format TCG translates guest instructions to an intermediate format which consists of simple machine code like instructions (called *micro operations*). This intermediate format is then translated into host instructions.
The indirect translation reduces the amount of platform dependent code as instructions from every supported CPU architecture are only translated to and translated from this format. Without this intermediate layer it would be necessary to be able to translate every instruction to instructions of every CPU architecture (see Figure 2.3) [23].

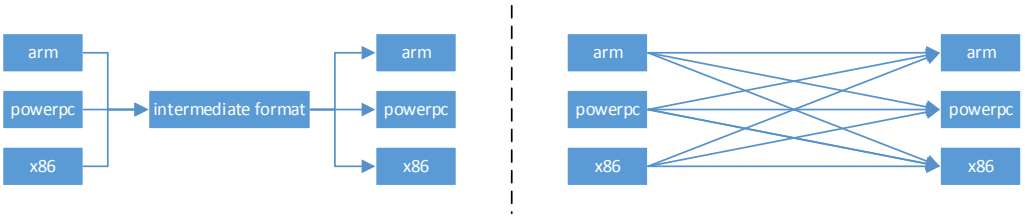

Figure 2.3: QEMU: Advantage of TCG's intermediate format; translation is only necessary to and from the intermediate format.

TCG Helper Functions As there are only very basic micro operations, it can be very difficult to implement the equivalent for complex guest instructions, e.g., the `sysenter` or `syscall` instruction on AMD64, by writing these micro operations directly. To ease

implementation of complex instructions TCG can insert a call to a TCG helper function at the corresponding position in the translation block. These helper functions are called during execution of the translated TBs. As TCG helper functions are normal C-functions it is much easier to implement complex instructions, however the execution is slightly slowed down because of the necessary function call [23].
The helper functions do not only allow easy implementation of complex instruction, but they are also an important tool to insert instrumentation code into the instruction stream, e.g., to run profiling code after each guest instruction.

Static CPU State In QEMU the static CPU state is the state, that TCG considers to be known at translation time. An example for this state is the IP. But depending on the simulated CPU this can also include more of the state, e.g., current addressing mode and parts of the status flags [5].
As all checks regarding the static state can be done during translation time, this assumption allows to generate simpler code. However, building TBs based on this static information, comes with some consequences:

1. Cached TBs must not only be selected by their starting IP, but additionally QEMU has to ensure their static CPU state is equal.

2. It is no longer sufficient to end a TB on a jump; the TCG also has to terminate a TB if an instruction may modify a part of the static CPU state and the resulting static state is unknown at the time of translation [5].

The overall translation process including the support for helper functions and the intermediate layer is depicted in Figure 2.4.

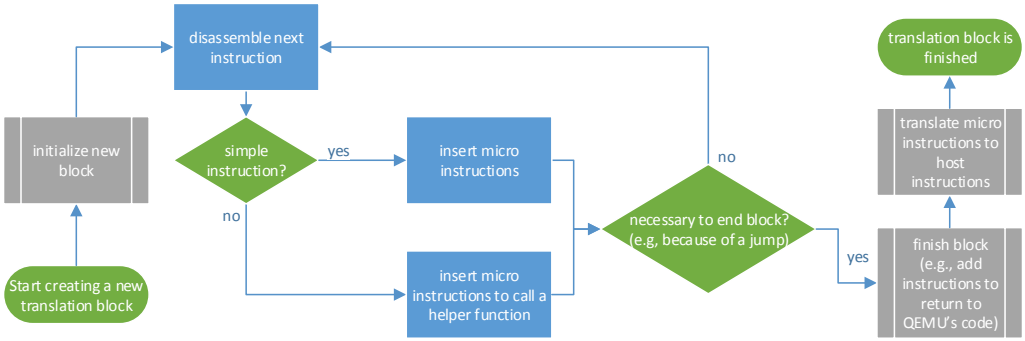

Figure 2.4: QEMU: TCG's process of translating guest code to a TB containing host code. Guest instructions are either translated to micro instructions or to a call to a helper function which simulates the effects of the guest instruction during the execution of the TB.

Execution of Translated Blocks

QEMU reduces the overhead for the translation of TBs by caching the translated blocks as it is already described in the basic approach (§ 2.1.1), but it also adds an additional technique called *TB chaining*. Other important details are the monitoring of guest code changes, the implementation to recover the guest IP for an interrupted TB and the implementation of multi-core support.

Translation Block Chaining As described in Section 2.1.1, TBs may contain at most one jump-like instruction; i.e., the execution of a common TB containing jumps with hard-coded addresses can only continue at two different addresses outside the TB.[2] The new IP can either be the last IP of the block incremented by one instruction, or the target of the jump.

Since searching for already translated TBs requires a time consuming lookup, its result is memorized in these TBs when the execution flow takes the respective path for the first time. Subsequent executions of the same jump can then be executed without the need for a lookup and the source TB can even be patched to contain a direct host jump instruction to the *chained* TB [23]. An example illustrating block chaining can be found in Figure 2.5.

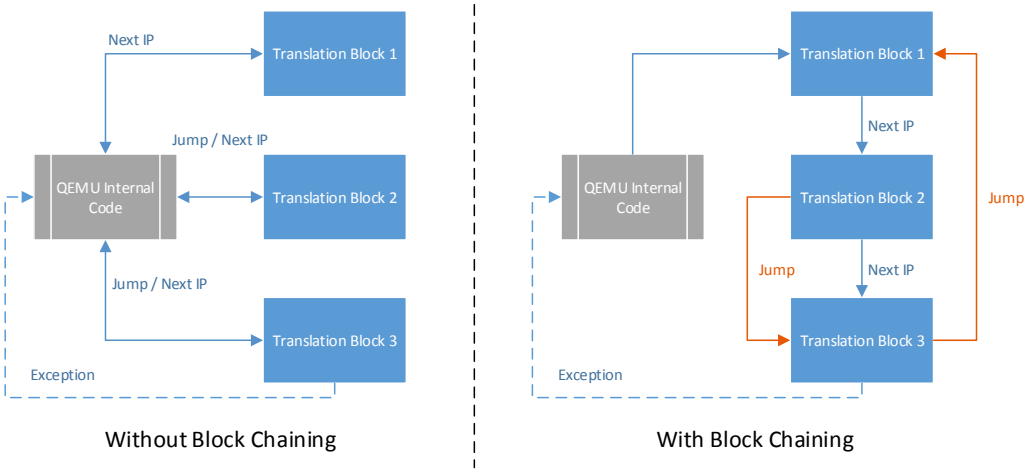

Figure 2.5: An example of a possible execution flow between several TBs, with and without TB chaining using direct host jump instructions. We assume the chaining is already set up.

The downside of the cached and chained TBs is the requirement to invalidate entries of this cache on code changes in the guest. If the TB chaining is done using host jump instructions, it is insufficient to invalidate the TB covering the changed guest instruction,

[2] This is not true for TBs, containing jumps whose target (guest) instruction pointer is not known, e.g., when jumping relative to a register's value. These TBs do not benefit from the described feature [5].

in addition QEMU must patch all TBs[3] pointing to the invalidated TB to avoid jumping in an invalidated block [5].

Monitoring Modification of Guest Code QEMU uses a guest platform independent approach to observe code modification by setting source pages for TBs to be write protected and monitoring the page faults afterwards. This approach is similar to the implementation of copy-on-write. The simulator keeps a linked list containing all TBs located in a write-protected page [5]. Once QEMU detects a modification to a code page, all linked TBs are invalidated. As mentioned before, we must also invalidate any links pointing from other TBs to this now invalidated TBs.

Guest Instruction Pointer Recovering When an exception or any other instruction, which forces QEMU to interrupt the execution of a TB, occurs, the simulator has to determine the current guest IP. As previously discussed, QEMU considers the IP to be part of the static state and hence its representation in the virtual CPU state is not always up-to-date within a TB.
QEMU solves this problem by rebuilding the current TB and temporary memorizing which guest instruction corresponds to which host addresses in the TB [5]. For instrumentation code inserted into a TB, this means that this code must stay the same as long as a TB may be rebuild, i.e., at least as long as a TB is executed. If the newly translated TB does not match the old one, the calculated guest IP might otherwise be wrong.
An optimization of QEMU is that it does not use a new buffer for the re-translation, but instead the old TB is overwritten and the translation is only done as far as necessary, i.e. it does often not translate the whole block. This approach is only valid under the assumption that the new and the old translation result is equal. When the result is different, e.g., because we did not longer emit a helper function for instrumentation, the resulting TB may consist of a mixture of the old and the new translation and is hence likely to be broken.

Multi-Processor Simulation To simulate multi processor systems QEMU simulates each processor in a round-robin procedure; the first virtual CPU is simulated by executing TBs for a certain time, then its state is saved and the second virtual CPU's state is loaded and the execution of the relevant TBs is started and so on. This avoids locking and synchronization problems, but it does not provide real parallelism [25].
PQEMU [14] is an approach for real parallel execution of several guest CPUs based on QEMU, however it is not yet included within QEMU's default code.

Accuracy and Determinism

QEMU is focused on speed and hence does not aim to simulate the correct timings of executed instructions. However, there is the optional *icount* [46] extension which

[3] While the amount of locations a TB can point to is limited to two, the number of TBs which can be chained to a single TB is unlimited.

implements a virtual instruction counter and uses it to execute a constant amount of guest instructions per virtual time unit, but this is still not equivalent to a cycle-accurate simulation.
In addition to this QEMU does not deliver interrupts precisely at the point in time they would be delivered in a real-world system, instead it is asynchronously periodically checked whether an interrupt was raised and if this is true, the chaining of the blocks is interrupted. The interrupt itself is handled once the execution flow returns from the currently executed TB [5].
QEMU processes asynchronous I/O operations which are requested by the guest code in a second thread [25] while the main thread continues the execution of TBs. As the duration of simulated I/O operations, e.g., copying data to a simulated hard drive, depends on the scheduling of the threads and other factors located in the host system, the point in time when an interrupt is raised to signal the competition is indeterministic and thus QEMU itself cannot provide a deterministic simulation.

As the simulated guest system is not completely isolated from the events in the host system, even an unrelated program inside the host may affect the exact course of events within the guest. In our case this is a drawback as we want to avoid modifying the execution of a guest system and we hence implement introspection techniques located outside of the guest. Because of the incomplete isolation, the guest system is still not completely independent of our approach residing within the host system.
However, the indeterministic behavior of QEMU can also be an advantage, as with determinism several executions of the same test would always result in the exact same guest code paths and hence they would only cover one of several realistic execution paths.

2.2 Introspection

In context of full system simulation and virtual computer systems, introspection is the process of accessing high-level information such as the content of certain variables used within a simulated guest OS or its applications from outside of the simulation. Knowledge on variables and structures used by the guest allow a better understanding of the guest's logical state and can be useful in a variety of situations like debugging or performance analysis.
Introspection is also very popular in the security research area, especially for digital forensic and intrusion detection applications. As a result there are a lot of different approaches, e.g., *Volatility* [55], *FATKit* [30] or *InSight* [48].

A fundamental problem of all introspection approaches is the *semantic gap*. Full system simulators are by design able to access every part of the guest's state, e.g., its register values and the memory content. This is insufficient, as they have no knowledge on the meaning of large parts of the contained data such as structures defined by the guest OS or programs. The interaction with the guest is done by simulating an ISA and the machine instructions issued by the guest only contain information necessary for the processor to

execute a program, but they are not sufficient to understand what is done on a high level. Figure 2.6 describes the interaction between the guest and the host.
An example: We assume the guest OS might save the number of currently running processes at the memory location `0xffffffff0`. The simulator is able to modify this value when the guest instructs it to do so by executing certain machine instructions, but it is only aware that the value of a location in the guest's RAM changed. As the simulator is unaware of the context of the executed guest machine instructions, it cannot know, that modifying this value indicates a change in the number of currently running processes (according to the guest OS's view).

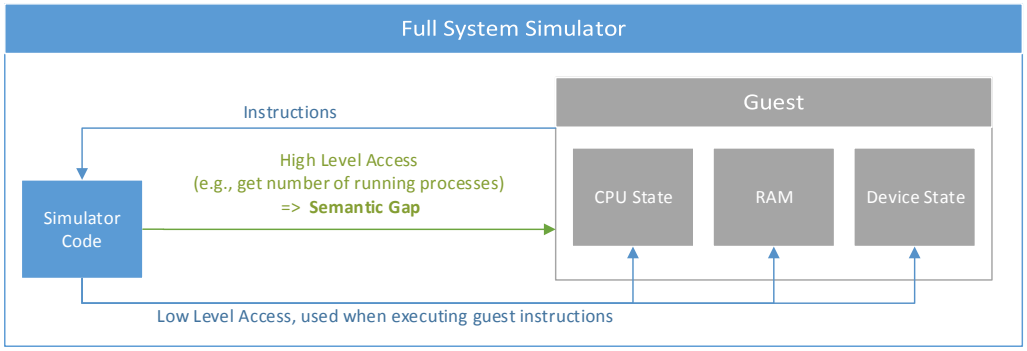

Figure 2.6: Interaction between the simulated guest machine and the host code within a simulator. The simulator can access every part of the guest's state, but it is unaware of the high-level semantics of this state which cannot be derived by using the ISA specification. In particular the semantics of most of the RAM's content can be defined freely by the OS and the individual applications and they are hence not easy to interpret.

We refer to the fact that the simulator is unaware of the context of huge parts of the guest's state as the semantic gap [10] and to solutions narrowing down the semantic gap as *semantic bridges*.

In this section we describe their common design ideas (§ 2.2.1), their individual ways of bridging the semantic gap (§ 2.2.2) and their vulnerability to countermeasures (§ 2.2.3). We only discuss the approaches for analyzing the state of the *guest OS*, to examine user programs, we can use very similar approaches once we have extracted the information from the guest OS, necessary to identify the memory of individual user processes.

2.2.1 Snapshot Based Introspection: Basic Idea

Current introspection approaches are mostly focused on forensic analysis or a similar scenario. As a result all programs we present here work on (memory) snapshots of computer systems, i.e., they do not monitor the change of the guest system continuously, but they analyze a snapshot of the system's state instead.

However, there are approaches which decide automatically when to analyze a snapshot of a running system, e.g., for intrusion detection systems this can be triggered by a write to a predefined memory region in the guest [21].
All but one of the approaches we discuss rely purely on RAM images. The advantage of this decision is, that these approaches are usable for memory dumps of normal (non-simulated) computers, too. The disadvantage is that the RAM content contains a lot but not all state of the current system, hence accessing information, e.g., located on swapped out pages or in registers of the CPU, is not possible.
As only few structures are predefined by the ISA such as the structure of page tables on X86, the OS is free to define the format and encoding of most information found in the RAM. Hence introspection approaches need to contain OS dependent code.

The idea of snapshot based introspection is to walk structures, e.g., the task control structures (`task_struct`) from Linux, located inside a memory dump and to extract the contained information. In case of the task control structures, it might be the process' name and its identifier [4]. To be able to access the structures and to follow pointers between them, introspection approaches must bridge the semantic gap by:

(A) Translating Virtual to Physical Addresses

The memory image of a guest represents the physical pages of the guest's RAM and hence we can only access the data within by using guest physical addresses. However, standard desktop and server OSes often run in a mode where they use virtual addresses themselves, hence their structures refer to each other by pointers to virtual addresses and the contiguous objects created by programs or by the kernel may be scattered in the guest physical memory [30]. An example can be found in Figure 2.7.

As the memory image represents the physical memory layout, we must be able to translate between these two address types to access such structures from the viewpoint of the OS, i.e., its virtual address space.

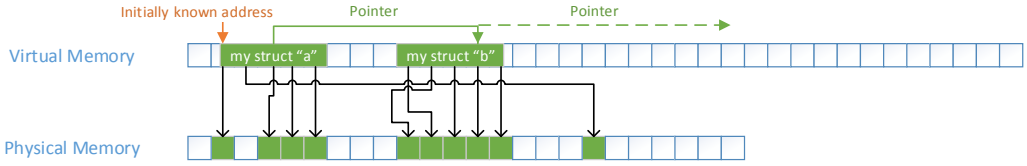

Figure 2.7: Linked structures and their location in virtual and physical memory. Objects which are contiguous in virtual memory can be scattered in the physical one.

(B) Obtaining Knowledge on the Representation of Structures in the RAM

Assuming we know the starting address of a structure, we still require information on the memory layout of the structure itself, to be able to access entries of this structure. See Figure 2.8 for an example layout.

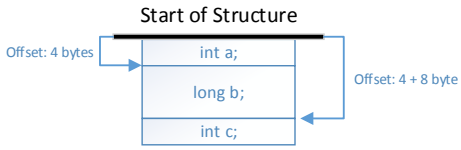

Figure 2.8: Example for the possible memory layout of a structure containing two 32 bit integers (a, c) and one 64 bit integer (b).

The required information is (i) the offsets from the start of the structure to the start of the individual entries, (ii) the size and format of the entires and (iii) the size of the structure[4].

(C) Obtaining Addresses of Single Instances of Important Structures

As the structures inside the memory are typically heavily interconnected by pointers (because they are organized in lists or trees), it is possible to find most structures starting from only a few memory locations. In the following we will refer to this locations as *starting points*.

These starting points must be somehow provided and then the trees and lists can be traversed using the information on the layout of the respective structures provided by item (B).

Structures for translation (A) are often predefined by the ISA of the computer system and we can provide a translation by re-implementing what the hardware does to translate an address. But it is still necessary to provide an initial location of the structure containing the information necessary for the translation, e.g., the root page table. Assuming the memory image is from a simulator, we can get this address from the simulated CPU or MMU.
The individual approaches differ in the way they accomplish the requirements (B) and (C), hence we present these methods in the next section (§ 2.2.2).

2.2.2 Snapshot Based Introspection: Existing Semantic Bridges

In this section we present the fundamental idea behind several "classical" approaches to bridge the semantic gap with focus on items (B) and (C) we defined in § 2.2.1. Some of these approaches pre-calculate the layout of necessary structures off-line, while others detect the layout of the structures during runtime. The last approach we discuss is rather new and does not explicitly solve the problems (A-C).

Off-Line Structure Extraction

There are three methods to obtain the necessary information on structures and starting points off-line. This information (often called *profile*) is only valid for the exact version of

[4]The overall size of a structure may differ from the sum of the sizes of its entries because of alignment reasons. If a padding is added to the end of a structure, the overall size of a structure cannot be determined by the offset and size of the last entry. However, a structure's size is important for analyzing an array of structures.

the OS binary it was created for. Handcrafting a profile is not reasonable for an universal approach, but the extraction based on recompilation or the usage of debugging symbols can bridge the semantic gap in several scenarios. Figure 2.9 and the following two sections describe these two off-line approaches.

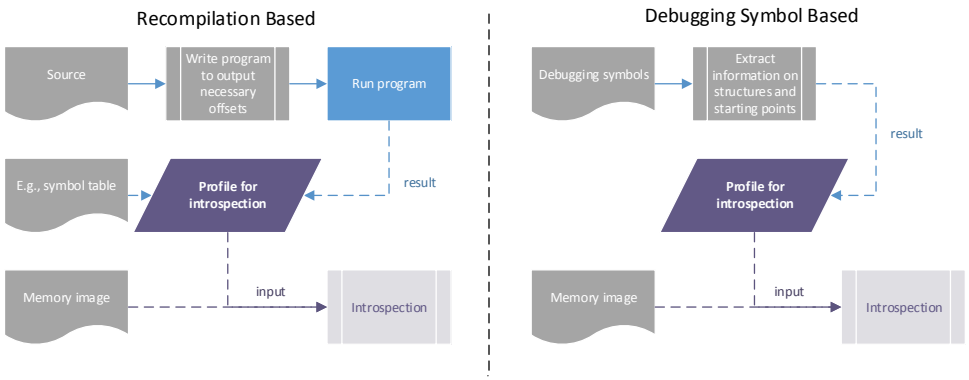

Figure 2.9: The approaches using off-line extraction create a profile which contains all necessary information on structures and starting points. This profile is then used during the introspection. The recompilation based approaches use modified source code built by the same compiler as the original executable file to print out sizes and offsets of structure entries. In general the starting points must be obtained using an additional source like a symbol table generated while compiling the original binary. The debugging symbol based approach however solely uses debugging information generated while compiling the original executable file. The symbols contain the layout of the structures and the addresses of global variables, the starting points.

Recompilation Based The memory layout of structures is defined by the compiler, thus it is possible to extract this layout by recompiling an executable's (modified) source code. The pre-requirements for this approach are the access to the parts of the source code which describe the structures and the ability to compile some source code with the same compiler and the same flags which were used to build the original binary.

Based on the definitions of structures found in the source code, it is possible to (automatically) generate the source for a program, which prints information on these structures [30]. In the C programming language, information on structures can be obtained by using the `offsetof(...)` macro and `sizeof(...)` operator [60].

The problem with recompiling modified programs is that we cannot assume, that the addresses of the global variables which we can use as starting points are equal to the positions in the original program. A solution for this problem is to use information memorized while compiling the original program. A symbol table for the executable, e.g., the `System.map` file for Linux kernels, is sufficient to solve this problem [30].

The *Forensic Analysis ToolKit (FATKit)* [30] is an example for a program suite for forensic analysis of a memory image which supports the described approach, but several other programs use this as a fall-back solution, too (e.g., *Volatilitux* [22]).

Debugging Symbol Based *InSight* [17] and *Red Hat Crash Utility* [3] are approaches which extract the layout of structures and the addresses of global variables (as starting points) from debugging symbols. Debugging symbols contain all structures used within a binary (detail are covered in § 2.3) and it is rather easy to extract this information.
An obvious limitation of this approach is that it requires debugging symbols of the currently running OS binary.

Automatic Structure Detection During Runtime

Volatilitux [22] uses an approach which is able to find starting points and the layout of structures by applying expert-knowledge on the OS internal layout and characteristics on the memory image. This expert-knowledge can reach from hard-coded numbers and offsets up to the normal runtime behavior of the OS.
To obtain an offset or a starting address of a structure, such approaches build a lot of assumptions, often by brute-forcing a range of possible values. These assumptions are then validated against the OS specific knowledge and possibly already determined offsets and starting addresses. An assumption regarding the value of an offset or an starting address can be considered to be correct as soon as all other possible values proved to be wrong. We provide a flowchart depicting the incremental extraction of profile information in Figure 2.10.

Example: How to find an initial `task_struct` in Linux

In a Linux system there is always one process called "swapper" with a hard-coded Process Identifier (PID) and its name is contained in its `task_struct` [22].

We scan the complete RAM byte per byte for this string and all memory locations which contain this string are considered to be possibly located in the searched `task_struct`.

Assuming we knew the offset between the name string and the PID in the `task_struct` we could filter out false positives: For every memory address at which we found the string "swapper" we could apply the offset and access the memory which represents the PID if this is a valid `task_struct`. Whenever the potential PID is not the known PID of the swapper process, we can ignore this occurrence as it cannot be the instance we search for.

However, this approach only sorts out locations which are definitely wrong, but there might be still several remaining possible locations. The number of this possible locations must be reduced further by using other checks until only one location remains, e.g., by using the fact that all `task_structs` are contained in a linked list. If we knew the offset to the pointers representing the list, we could

traverse this list and check whether we return to our start point or whether we end up trying to access invalid memory.

The problem with the described approach is that most offsets are not stable over different builds of the Linux kernel, so we cannot hard-code them. What we know is that the size of the `task_struct` is normally within certain limits, thus we can run the whole test several times with different values for the necessary offsets. Whenever a test with certain offsets results in rejecting all locations where we found the string “swapper”, we know that at least on of the offsets is wrong.

When the implemented tests are complex enough, it is very likely, that we end up with a single combination of used offsets and a location for the swapper’s `task_struct` which passes all tests. In this case we found an initial `task_struct` and some offsets for entries within every `task_struct`.

In contrast to both approaches described before, this one does not require additional files to analyze a memory dump. As it does not require the exact offsets within structures, this approach is rather insensitive to different OS versions.

The disadvantages of automatic structure detection is the complexity of writing and extending such an approach and the risk of detecting wrong offsets in case a wrong assumption withstands all checks against the expert-knowledge.

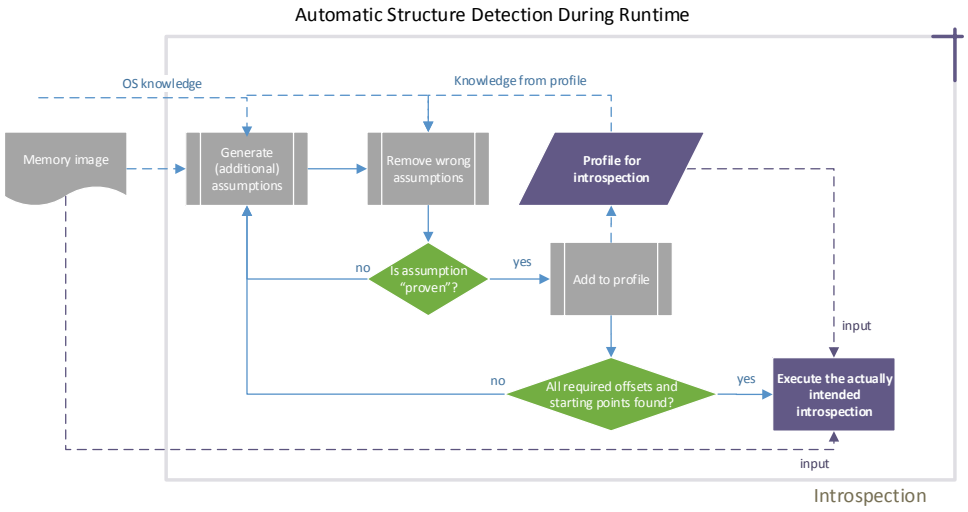

Figure 2.10: Automatic structure detection during runtime based approaches are not separated into profile generation and introspection, but instead they acquire the profile gradually during the introspection itself. Generic knowledge on the OS and the already acquired profile information is used to guess and verify not yet known parts of the profile.

Example Based Introspection

Virtuoso [17] is a tool differing from the approaches described before by not requiring any explicit information on the OS internal structure. This approach generates introspection code automatically by recording system wide traces while executing example programs which request the desired information from the target OS by using its Application Programming Interface (API). Figure 2.11 illustrates the overall workflow.
To generate executable code, the recorded traces are analyzed automatically and reduced to instructions necessary to access the desired information. As this code replays the relevant parts of the instructions which happened within the guest system, it requires to read and write CPU registers and the RAM. Because writing to the guest machine would modify the guest's state, Virtuoso provides a copy-on-write environment to avoid tampering with the guest RAM and registers.
Virtuoso's approach is a promising way to easily provide introspection code for information exposed to the userland, but as it generates code based on examples, the generated code cannot be guaranteed to cover all possible cases [17]. Furthermore, we assume it is impossible to extract internal information such as private variables of the kernel which cannot be requested by an example program.

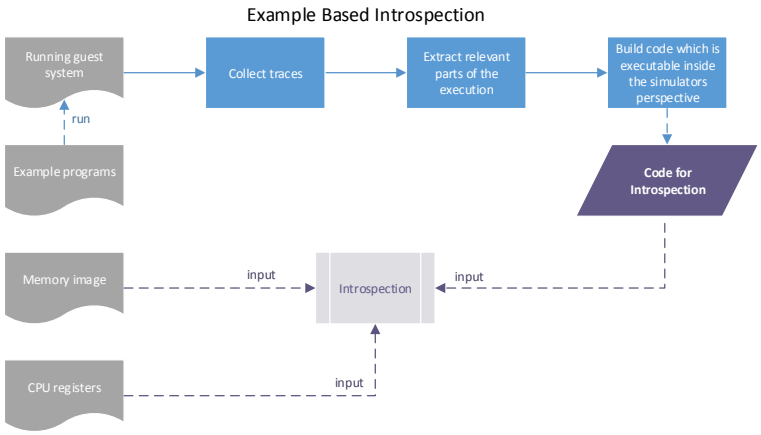

Figure 2.11: Example based introspection collects traces of test programs which run within the guest and use API functions providing information, we want to be able to access from the outside, too. These traces are automatically analyzed and the relevant parts are translated to code runnable on the host system. Inspecting the state of the guest is done by executing this code.

2.2.3 Possible Countermeasures against Introspection

All introduced introspection approaches rely on certain assumptions on the layout and content of structures used by the OS. A malicious attacker can utilize this by breaking these assumptions via modifying the kernel structures during runtime.
An exploit for this weakness is *Direct Kernel Structure Manipulation* [4]. This approach

is able to prevent introspection approaches from working correctly and even to trick them into presenting data which seems to be valid, but which is in fact wrong.
To achieve this, the exploit modifies the kernel structure entries seen by the introspection approach and tricks the kernel code into not using these modified parts, but to use a correct copy located somewhere else. This results in two sets of independent data, one accessed by the introspection tools and one used by the kernel code.
An attacker can change the kernel code behavior by either patching the code during runtime or by using more advanced methods (for details refer to the paper describing this attack [4]).

2.3 Debugging Symbol File Formats

Debugging symbols describe the relation between the machine code generated by a compiler and the source code. They are the base upon which debuggers operate, but as we previously discussed, they can also be used to bridge the semantic gap (§ 2.2.2).
The de facto standard format for Linux and Unix is the *DWARF* [18] format. DWARF is often used in combination with the *Executable and Linkable Format (ELF)* [57]. *STABS* [41] is a nowadays seldom used debugging symbol file format which was once commonly used in Unix and Linux. On Microsoft Windows, there is the *CodeView* debugging symbol file format [58] which is often contained within the old *DBG* or the new *Program Database (PDB)* file format [49].
The following sections discuss DWARF (§ 2.3.1) and present some basic facts on the CodeView format embedded in PDB files (§ 2.3.2).

2.3.1 DWARF

DWARF exists in different versions and we discuss the most recent version 4 (2010), but most information we present is also valid for versions 2 and 3. The DWARF standard itself is rather extensive and hence we only provide a basic overview [18].
DWARF splits the represented information into *Compilation Units*. They typically represent the debugging information for an individual object file, which all together form the binary file the symbols where created for [18]. In context of the C language, an object file is normally created by compiling a single C-file and all the included header files.
Within a Compilation Unit, the format structures most debugging information as a tree (see Figure 2.12). A function is for example represented as a node with children representing the function's parameters and contained variables [18]. The single nodes of this tree are *Debugging Information Entries (DIEs)* which have a certain type, e.g., `DW_TAG_subprogram` or `DW_TAG_variable`. According to their individual type they have different optional and required attributes such as the byte size of a variable (`DW_AT_byte_size`) or its name (`DW_AT_name`). DIEs representing functions contain — among others — information on the address range, which corresponds to the code of the functions.
DWARF tries to move information used by several instances into shared structures, for example the size and encoding is not directly contained in the DIEs representing a

parameter or variable. Instead this information is encoded in a shared DIE, e.g., in case of a simple variable such as an integer, this is a DIE of type `DW_TAG_base_type`. To represent this relation, the DIEs can refer to other DIEs with attributes such as `DW_AT_type`.

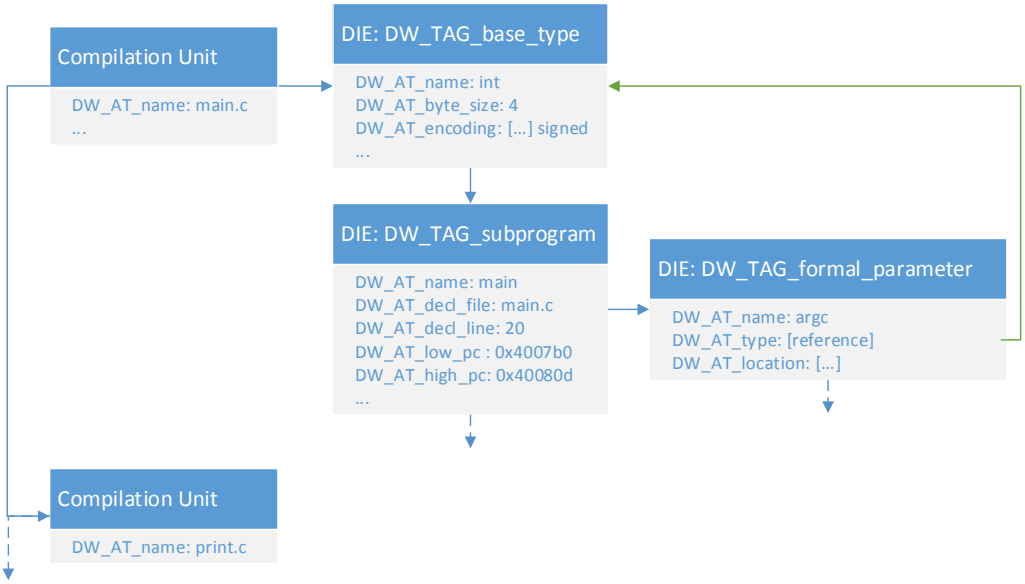

Figure 2.12: A fragment from a simplified DWARF tree representing a binary file. Each DIE can only point to one child and to one successor. Several children of one DIE are represented by a pointer to the first child, whose successor is the next child and so on. Additional other DIEs can be referenced by attributes, e.g., `DW_AT_TYPE`.

The basic requirements for a debugger are the association of IP values with code locations and the access of variables' values during runtime. The debugger can extract the first by using *line number information* and the later by analyzing *location descriptions* and *type information*.

Line Number Information

Line number information is the data necessary to link machine code addresses to source code positions and vise versa. The information provided by DWARF is the IP, the assigned source file, the line number and some additional hints, e.g., whether the IP is a recommended location for a breakpoint [18]. The provided granularity depends on the compiler and the structure of the source file, in particular not every line number needs to be contained in this table as a line might be empty or not result in any generated instructions.
If this information was memorized as a table sorted by the IP value, it would possibly be very huge as every instruction address in the binary results in an own line of the table.

DWARF uses two techniques to reduce the necessary size. The first step is to remove all lines which only differ in their IP value from the previous line. Because of the sorting, the remaining information is still sufficient to assign an IP value to a source code location. The second step is to encode the information of the remaining table using a *line number program* written in a byte-coded language. To reconstruct the table, the user must execute this byte code on a state machine described in the DWARF specification. The state of the machine represents the content of a single line such as the IP value or the line number. Therefore it is not necessary to encode all information contained in each line, but instead only the difference between the individual lines must be represented by the byte code.

Location Descriptions

Assuming a debugger wants to access the value of a variable or function parameter during the runtime of a program, it requires information on how to do so. The required prior knowledge is the current IP and the DIE representing the variable to be accessed. If we know the IP, we can find the DIE by using the line number information and by walking the tree structure.
This DIE contains an entry called `DW_AT_location` which encodes the instructions necessary to access its value using a *DWARF expression*. In general this is not a single expression, but a list of such expressions; each entry being associated with a range of IP values in which the DWARF expression is valid.
DWARF expressions are streams of opcodes which must be executed on a stack machine. There is a huge amount of instructions reaching from pushing constant values to the stack over reading the memory content at a calculated position to shift operations [18].
Only few variables are directly accessible by a simple name within the source code of a function, a lot of variable are indirectly accessed by resolving pointers or accessing members of structures. In a function for example, there might be only a variable name for a pointer to an integer, but not for the integer itself. In this case only the DIE for the pointer is a child of the function's DIE. However, we are able to derive from the pointer's DIE that it is a pointer and where we can find the DIE representing the pointers target. The DWARF DIEs for this example are depicted in Figure 2.13.
To retrieve the value of this integer we must first execute the DWARF expression of the pointer and afterwards the expression found in the integer's DIE.

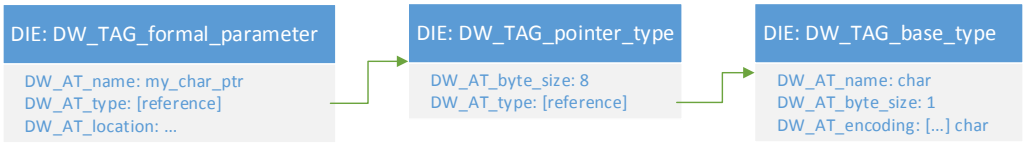

Figure 2.13: Example for DWARF DIEs representing a function parameter which is a pointer to a `char`. The DWARF tree structure is omitted; only the references between the relevant DIEs are depicted.

Type Information

To be able to interpret memory or register values, the debugger must know several details on the variable he expects to find, e.g., the endianness, the byte size and the encoding. This information can be contained in the DIE belonging to the individual variable or the current compilation unit. For some attributes like the endianness it is not necessary to specify them at all; in this case the platform defaults are used [18]. Other values like the byte size are pertaining to a certain type and are hence located in the type's DIE.

2.3.2 CodeView Symbols in Program Database Files

Debug information created by Microsoft's tools such as Visual Studio are normally contained in Program Database (PDB) files. The PDB file format is not officially documented [49], however, there have been attempts to reverse engineer the file format and hence some details on how this file is structured and what debugging format is used are known [49].
To implement a debugger, using a library provided by Microsoft is intended. The *DbgHelp* library [12] and the *Debug Interface Access (DIA) SDK* [13] are available for this purpose.

Program Database File Format

A rather old version of this format, used in Microsoft Windows 2000, was partly analyzed by Schreiber [49] and an implementation of a parser for newer files can be found in the *pdbparse* project [15].
The PDB file format consists of several streams describing the debugging information. One of them is the *Global Symbol Stream*, which contains offsets and types of global symbols. However, it can be necessary to translate these offsets using the optional *OMAP Stream* [16].
OMAP is a feature used in Microsoft's build tools to move blocks of code within the binary, presumably to optimize the execution of the application. These modifications are not directly recorded in the relevant debugging information, hence the translation using the *OMAP Stream* is necessary [49].
Some other streams like the *Type Stream*, containing information on the data structures used within the program, are also understood [16].

CodeView Format

The symbol information in the PDB file is encoded using the CodeView format [49]. We describe this format as it was documented 1993 in *Tool Interface Standard (TIS) Formats Specification for Windows* [58] as we do not have access to a newer version, hence some of this information might be outdated by now.
The CodeView format basically consists of two different tables, one containing the type information (called *$$TYPES*) for variables and one representing the symbols in the binary file (called *$$SYMBOLS*). The tables themselves contain records of variable length. Addresses referring to locations in the binary are encoded by naming their logical segment, they are contained in, and the offset within it.

The nested structure of program code is represented by including records which indicate that a new level is opened, e.g., a function start symbol, or closed.

$$TYPES This table contains information on types which are used in the $$SYMBOLS table. There are predefined record types, e.g., for integers and pointers, but it is also possible to add custom types.
There are special records for classes, arrays and structures which can contain other records with information on the members of this types.

$$SYMBOLS The $$SYMBOLS table describes the structure of the program and in particular contains information on how to access the value of variables and parameters. This information is normally encoded using predefined symbols, e.g., with a record type for a base pointer relative location; the parameters for this record are the offset relative to the base pointer register, the variables name and a pointer to its type.
Nevertheless, the symbol format was designed to allow specification of expressions which are executed on a stack machine in order to compute addresses for symbols' data, but this feature was not used by Microsoft in 1993 [58].

2.4 Used Programs and Libraries

We used several programs during the research and the implementation of our approach. *GNU Debugger* [20] was very useful when we searched for bugs within our debugging symbol interpretation and to evaluate the suitability of a debugger for introspection. *Libdwarf* [24] and the included *dwarfdump* program facilitated the interpretation of the DWARF debugging file format.

2.4.1 GNU Debugger (GDB)

GDB is a very popular debugger with support for several source code languages including C and C++. It is available for Unix, Linux and Microsoft Windows.
Besides debugging on the local machine, GDB supports remote debugging [20], which allows debugging on different computer, possibly with a different ISA. This feature can be used in combination with the GDB server stub, e.g., contained in QEMU (§ 2.1.3).
GDB supports *tracepoints* [50, 54] for remote targets. They are designed to collect data of a program with low overhead during runtime. The user can provide instructions on which data should be collected when reaching certain instructions within a running program. This feature allows collecting data into a buffer with lower impact on the timings of the program. In order to minimize the impact of the data acquisition, QEMU pre-calculates the instructions which are necessary to access the requested variables before the beginning of the tracing.
These instructions are called *agent expressions* and they are evaluated directly on the target. If for example a user requests the value of a certain variable, the debugger must normally access the debugging symbols, parse them and extract the necessary operations. To avoid this lookup on every hit of a tracepoint, it is only performed once and the result

is memorized as an agent expression. These expressions describe what should be done on the target, e.g., to access the value of a variable. They are implemented as a byte-code interpreted by a stack machine on the target [54].
The interpreter has no knowledge on debugging symbols and contains only instructions on a machine level like `add` or `mul`. To access the memory of the debugged program, instructions like `ref32` are available. This instruction reads 4 bytes at the address, contained at the topmost stack frame, and pushes the result to the stack.
Another benefit of this approach is that it avoids the need for a debugging symbol parser on the target.
However, agent expressions are not available for every target, in particular the server stub of QEMU does not support this feature currently.

2.4.2 Libdwarf and Dwarfdump

The libdwarf library was initially developed by Silicon Graphics and is released under the LGPL. It provides a more abstract view on the DWARF structures, e.g., the library can execute the line number programs and provide the resulting table (see § 2.3.1), and it performs the parsing of the binary debug information into C++ structures [24].
In addition to the library there is a utility to print the information contained in a DWARF file. It is called dwarfdump and uses libdwarf itself.

2.5 Terms and Definitions

Within this thesis we use several commonly known terms and acronyms. Although we introduce them within the text, they can also be found in the glossary at page 91. In addition to these we introduce two own terms to avoid impreciseness when referring to modules. Table 2.1 defines the two terms.

Term	Description
Introspection Module (I-module)	A module used in our introspection environment
Kernel Module (K-module)	A modules which can be loaded into an OS kernel

Table 2.1: Important terms and definitions used in this thesis.

Chapter 3

Analysis

Full system simulators can simulate a computer system on a very high level of detail and all of its detailed state can be observed — in contrast to a real-world system. Hence full system simulators are useful tools for measuring the runtime behavior of a computer. While it is easy to measure hardware events like cache misses or mis-predicted instructions, it is often also relevant to measure high-level semantic information on the state of the OS and the programs.
The methods currently used to collect such high-level information like inserting hypercalls into the OS have several drawbacks, e.g., they modify the result of the measurement because they affect the execution flow within the guest. In this chapter we aim to discuss the limitation of existent approaches and develop ideas for alternatives.

An example for a full system simulator based measurement is the analysis of the statistical properties of duplicated pages in a Linux system from Gröninger [25]. One important property of such pages is their origin, e.g., whether they are used by the OS or by programs. This information is contained within the structures of the OS and cannot be directly derived from the hardware state.
The kernel used in the benchmark was modified to inform the simulator on certain events which were relevant for the measurement, e.g., when a task switch occurred or on certain changes within the buddy allocator of the kernel.
In order to estimate which quantity of such events might be necessary we asked the author for some statistics on the number of events. We depicted the four most often issued events in Figure 3.1. *DispatchEvents* indicate the scheduling of a different task, *BuddyAllocationEvents* refer to changes in the state of the kernels buddy allocator, *PageCacheEvents* provide information on pages in the page cache and *VMAMapEvents* collect data on mappings between virtual and physical memory. The detailed descriptions of the workloads used in the benchmarks can be found in the related thesis [25]. Though there was a case where the amount of events was only about 24 per second, there were normally several hundreds of events. The worst case was about 8537 events per second for the SPEC 433 benchmark.

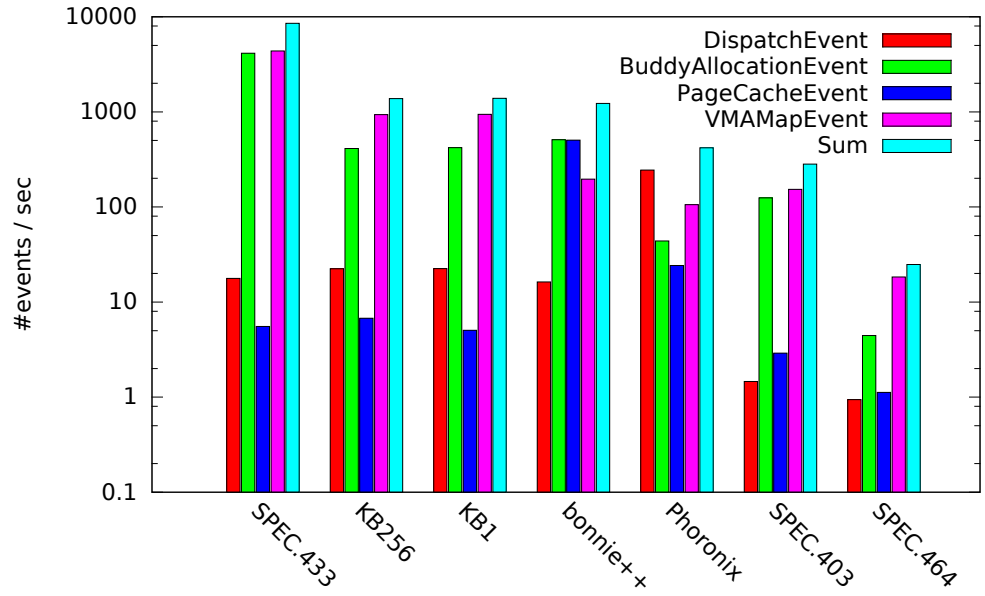

Figure 3.1: Average number of hypercalls per second issued by the implementation in thesis [25]. Only the four most often issued events are displayed — totally 24 were recorded. "Sum" only refers to the sum of this four events.

The existing introspection approaches (§ 2.2) which can provide high-level information on the system state have several drawbacks in scenarios such as the one described above. First of all, they are snapshot based and extract all requested information at once from a guest system frozen for a short time. As the high-level information is possibly required on the same level of detail as the low level machine information, these approaches would need to be run after each memory modifying guest instruction. In fact it would be sufficient to run them only if a part of the extracted state changed, but these approaches can only tell if something changed, when they have already analyzed the state.

The increase in runtime when running an analysis of the guest after each executed write instruction would not be acceptable as for example in the benchmarks of Gröninger [25], about three millions of memory writes per second occured in average. Another problem with snapshot based introspection is that an accessed structure might be partly written while the introspection is executed and hence the introspection may provide wrong results or no results at all.

The solution for both of this issues is to use an *event-based approach*. Instead of polling for a change of the internal state, it would be advantageous to be informed when something changed and when it is save to read the structures which are to be accessed. However, no introspection approach allows that currently. To implement such functionality it is therefore only possible to modify the guest system, e.g., by modifying the source code [25] and recompiling it, to inform the simulator of a changed state. When using such

an approach, the requested information can be directly passed to the simulator, without using real introspection at all.

To ease the discusson of problems related to event-based approaches we provide a more detailed view on the semantic gap in Section 3.1. Afterwards we describe some possible approaches in Section 3.2 and discuss the problems when utilizing a debugger for introspection in Section 3.3. This chapter ends with Section 3.4 which covers the considerations on the basic design of an event-driven introspection approach which overcomes the issues of the previously described approaches.

3.1 The Semantic Gaps

To understand the issues which have to be solved to implement event-based introspection we divide the semantic gap into *two* different gaps:

Gap 1: What is the *current context*?
(Example: Which program (or the OS) is currently running?)

Gap 2: How to *access data* within this context?
(Example: How to get the value of variable `argv`?)

The first gap is in particular relevant if the target of introspection is the state of programs within a simulated system. The different programs normally use the same virtual address ranges, therefore an access to a certain virtual address can only be associated to a certain semantic meaning if we know which process is currently running and hence whose virtual address space is currently used. Figure 3.2 depicts both gaps and we use it as a basis to depict the semantic bridges in the following sections.

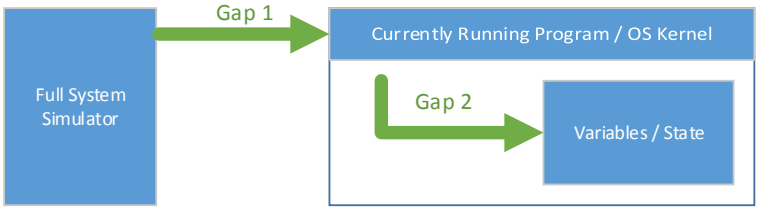

Figure 3.2: Detailed view of the semantic gap. The first gap describes the problem of determining the current context, e.g., which program is currently running on the CPU. The second gap represents the problem of knowing where and how to find specific parts of the guest's state such as the values of variables.

3.2 Approaches for Event-Based Introspection

There are several different possibilities to extract the state of a computer system, the most basic approaches are however no real introspection approaches, but instead they modify

the guest machine, to actively communicate the required information to the simulator. A simple approach is to modify the source code of the guest OS and of all guest applications which are of interest.
There are more sophisticated methods which allow instrumentation of the kernel or programs without recompiling, however they are normally not intended to be used to expose information directly to a simulator and hence it would be necessary to extend them with this feature. An example for kernel instrumentation was presented by Tamches and Miller [56]. They developed a tool called *KernInst* which allows to insert self-written code into a Solaris kernel during runtime. For normal programs there are a variety of approaches, static binary instrumentation based ones like *ATOM* [52] or *PEBIL* [31] and dynamic binary instrumentation based ones like *Pin* [36] or *Valgrind* [42].
These tools for binary instrumentation and modification soften some disadvantages of the source code modification approach, e.g., they do not require a recompilation of the binary, but their basic approach is similar: They rely on changes within the guest. For this reason we cover all these approaches together in Section 3.2.1.
The following two sections (§ 3.2.2 and § 3.2.3) describe the ideas on how to avoid modification of the guest and to use real modification free event based introspection.

3.2.1 Source Code or Binary Modification

Within the context of a program its state can easily be obtained by inserting instructions to expose the interesting values at certain locations. The source of the Linux kernel for example can be modified to print every process switch by inserting a `kprintf(...)` instruction into the code used to switch processes. To transport the information up into the simulator, a hypercall or a magic instruction (a normally invalid instruction on an ISA) can be used [25].
The semantic gaps are bridged implicitly, by using instructions which are executed in the guest's context ("local agent"). Gap 1 can be bridged by instrumenting the OS to announce every context switch or by appending the current context (which is known to the running programs and the kernel) to every hypercall. Gap 2 is automatically bridged as the code we insert runs within the guest and hence there is no semantic gap from its perspective. Figure 3.3 depicts this solution.

Characteristics

The advantage of the described approach is — at least for the source modification — that it is comparatively easy to implement. It suffers however from a multitude of disadvantages, as it relies on modifying either the source code or the executed binary instructions and hence it always modifies the behavior of the guest system. This has several unwanted consequences:

- This approach can be detected by the instrumented programs, e.g., because of changed timings.
- The measurement itself impacts the results.

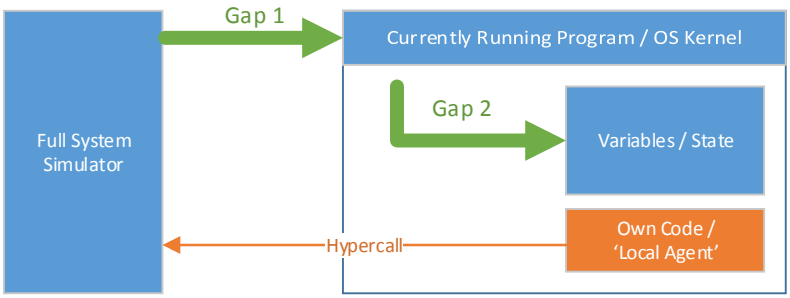

Figure 3.3: Semantic bridging by source code or binary modification. The semantic gap is closed implicitly by inserting code into the guest which passes the relevant data to the simulator.

For source code based modification, the complete source code and the build chain must be available and, as for statically modified binaries, the instrumentation can only be done off-line, i.e., the events which are to be recorded cannot be changed during the runtime of the OS or a program. In addition to that the patches to the source code must be adapted for every different version of a program which should be instrumented.

A further disadvantage of this approach is that it does not use unmodified standard binary files, even with dynamical binary instrumentation they are modified within memory. Using unmodified binary files on an unmodified guest, however, would allow to measure a workload on different levels of detail without modifying the execution and hence yield results of better comparability. When using a deterministic full system simulator the execution flow would be even exactly equal.

3.2.2 Connecting Debuggers to Programs and to the Operating System

In order to improve the previous approach debuggers can be used, which connect to unmodified programs and the OS. This is depicted in Figure 3.4. As debuggers work by setting breakpoints and by waiting for these to occur, they also work on an event base.

The debuggers for programs run within the simulated system and hence there is still a "local agent". To solve the second semantic gap, the debuggers can add breakpoints at code locations and use the debugging symbols to access the value of variables. As the debuggers are aware of which program they debug, they can pass this information together with the extracted data to the simulator and hence solve the first gap.

The kernel itself must be debugged from the outside, i.e., the kernel debugger cannot run within the guest system. This can be done by using features like remote debugging in combination with a debug stub in the simulator (see § 2.4.1 and § 2.1.3).

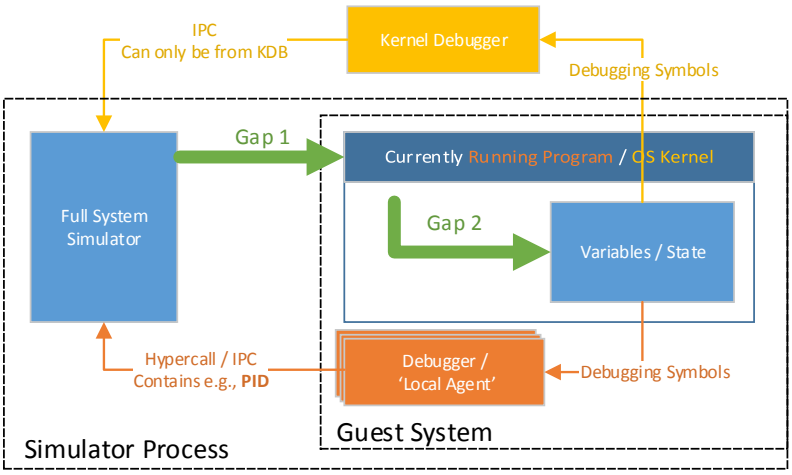

Figure 3.4: Semantic bridging by using debuggers which run inside the guest to debug the guest's applications and a debugger for the kernel which is located outside of the guest system.

Characteristics

This approach presents a solution easy to use. There is no need for a modification of the binaries or the source[1]. When debugging symbols are available it is possible to use standard binaries.

However, this approach is still detectable by programs and in particular it modifies the timings and the state of the guest system heavily by starting additional debuggers within it. In addition to that, even for a small number of breakpoints this approach becomes very slow. We evaluated a similar approach (details in § 3.3) and the simulation speed decreased by the factor 2.8 to 9.0 even though we monitored less than 10 events per second.

3.2.3 Debugging Completely from Outside the Simulated System

A next step is to remove all "local agents" from the guest as they always imply modifications to the guest's behavior. To achieve this it is necessary to give up all debuggers which run inside the guest. When using only the externally attached debugger as shown in Figure 3.5, the first semantic gap is no longer implicitly closed. Only a single debugger is connected to the guest machine regardless of which program is currently running, hence the debugger does not represent a certain context anymore.

[1] However, debuggers like GDB do actually modify the binary instructions within the RAM if software breakpoints are used [19]. But when we use hardware breakpoints or breakpoints provided by the debugger stub of a full system simulator, this is not necessary.

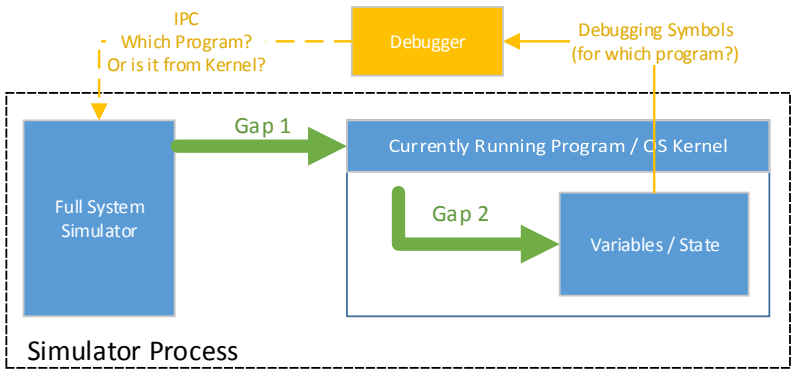

Figure 3.5: Semantic bridging by using a debugger outside of the guest. The first semantic gap is not bridged.

Characteristics

The differences to the previous approach are that it does not modify the timings within the guest system, but on the other hand it also does not bridge the complete semantic gap. *The first semantic gap remains unbridged.*
Using such an approach is often possible for instrumenting an OS kernel as its code and data are normally mapped into every virtual address space at the same position. Hence it is possible to tell solely by the IP whether the CPU is currently executing code of the OS. This is not true for normal applications and hence this approach does not support instrumentation of applications within the simulation.
In summary, this is still not a satisfactory solution because of the only partially bridged semantic gap and the slow performance inherited from the usage of a debugger.

3.3 Problems when using Debuggers

As mentioned before, we tested the feasibility of using a debugger for the instrumentation of simulated systems. We implemented the approach using only an external debugger connected to the simulator (§ 3.2.3).
We utilized GDB by using its text message based *machine interface* to communicate with the debugger. This interface is the intended way to communicate with GDB when building an application which uses GDB's features. GDB connected to QEMU in which we ran an unmodified Ubuntu. Table 3.1 contains details on the host and on the guest system configuration.
We decided to track three important events: task creation, deletion and switch. For creation and deletion we recorded the process identifier and the name, in case of a task switch we only recorded the identifier of the process we are switching to. The exact breakpoint locations and the access variables can be found in Table 3.2.
We measured the execution time of the compilation of a minimal Linux kernel, in one test case with only one thread (j1) and in the other one with two threads (j2). The runtime we specify is the mean of 5 executions of this workload.

Property	Value
Host OS	Ubuntu, Kernel 3.2.0-52-generic
Host Hardware	Intel Q6600 @ 2.40 (64 bit), 4 cores GHz, 7 GiB RAM
Host GDB	GNU GDB (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Guest OS	Ubuntu, Kernel 3.5.0-23-generic
Guest Hardware	AMD64, 1 core, 1024 MiB RAM

Table 3.1: Details on the host system running QEMU and on the simulated guest system.

Breakpoint Location	Accessed Data
start of function `wake_up_new_task`	`p->pid`, `p->comm`
start of function `exit_notify`	`tsk->pid`, `tsk->comm`
start of function `__switch_to`	`next_p->pid`

Table 3.2: Location of the breakpoints and the variables we access on each occurrence.

Our measurements (see Table 3.3) indicate a significant slow down by factor 2.8 to 9 even though GDB only handled below 10 events per second. Because of this surprisingly high performance impact we measured the duration of requests to read a value of a variable using GDB. This duration was relatively constant 0.04 seconds for each read operation, e.g., to read the process identifier. Assuming we only want to read a single value at each breakpoint, this results in a theoretical limit of about 23 events per second. This is much to slow to be useful for measurements which can easily lead to several thousand events per second.

		Runtime (s)	Runtime Factor	#Events/s
QEMU,	j1, icount=2	$\simeq 4634.0$	1.0	—
QEMU + GDB,	j1, icount=2	$\simeq 12991.2$	$\simeq 2.8$	$\simeq 7.07$
QEMU,	j2, icount=2	$\simeq 4828.8$	1.0	—
QEMU + GDB,	j2, icount=2	$\simeq 43584.2$	$\simeq 9.0$	$\simeq 8.79$

Table 3.3: Measurement results, each test was run 5 times and the mean values are given. The time is measured in host time, i.e., wall-clock time.

In order to understand the reasons for this extreme slowdown, we observed GDB itself while it was running and found that a lot of time was spent in functions which are responsible for parsing and searching in debugging symbols. As our Linux kernel including the debugging symbols is about 150 MiB in size, spending a lot of time for searching within these symbols seems to be realistic.

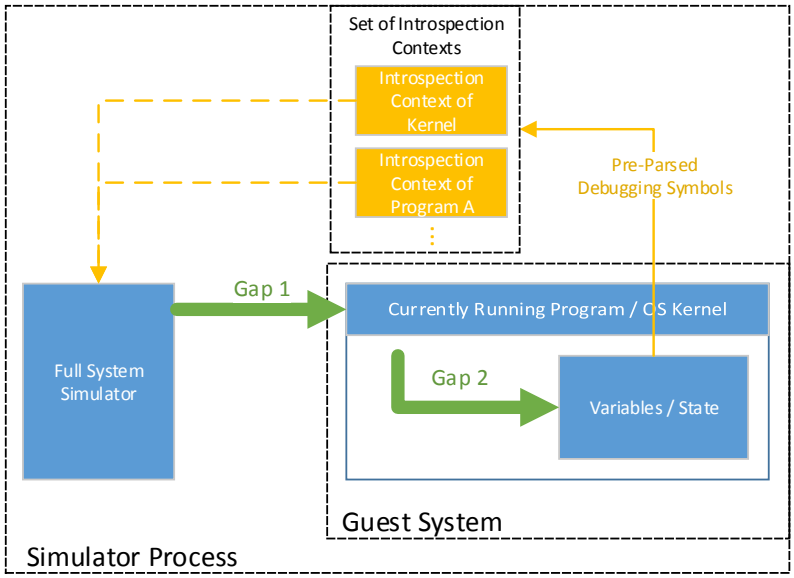

Figure 3.6: Semantic bridging by sets of introspection contexts. The set of introspection contexts is selected by the introspection context for the kernel. The kernel's context is always part of the current introspection context set assuming the kernel is represented in all virtual address spaces.

3.4 Resulting Approach

As we discussed in § 3.2 all solutions have important drawbacks. The last approach we presented still lacks two important features:

1. A bridge for the first semantic gap.
2. A sufficient execution speed of the simulation.

We therefore propose a further approach for event based introspection which shall overcome these issues. Figure 3.6 describes its semantic bridges. The important changes to the single debugger based approach are:

The Replacement of the Debugger We observed in our experiments (§ 3.3) that the overhead of a debugger was much to high to be able to handle several thousand events per second. We identified the continuous parsing of the debugging symbols as an important reason for this overhead and hence it should be replaced with something much faster.

Instead of parsing the debugging symbols during runtime it is better to pre-calculate the necessary instructions and directly create C code which extracts values of variables in the guest system during runtime. Additionally it is advisable to avoid complex communication methods like sending text messages when using GDB. To

minimize this communication overhead, the debugging code should be located within the address space of the simulator.

The Introduction of a Context Aware Introspection If introspecting single guest applications, their breakpoints for locations within the virtual address space must only be used when the CPU currently uses *their* address space. Therefore a mechanism is necessary to decide which events, i.e., breakpoints, are used within a certain context.

There are several different logical execution contexts within a single virtual address space such as the kernel, a program and libraries. To represent this, the current *introspection context* must consist of a set of contexts.

The Construction of Operating System Dependent Semantic Bridges To be aware of the current context of the guest, it is necessary to bridge the first semantic gap. As mentioned before, the kernel is normally located within every address space at the same place, hence the breakpoints for the kernel can be used regardless of the current context. The kernel on the other hand decides when to switch between different contexts, e.g., when to switch from one process to another. Thus the current context can be determined by tracking important OS events such as process creation, deletion and the process switch.

3.5 Conclusion

High-level information on the system state in full system simulator based measurements is necessary to understand the reasons for and meaning of low-level information. Current sources for high-level information which provide a high performance rely on modification of the guest system and hence modify the measurement. Snapshot-based introspection approaches are not feasible for continuous measurements because of their huge overhead. Therefore an event-driven introspection approach is necessary which allows to introspect the kernel as well as userland programs without impacting the simulators performance excessively. Because of multi-tasking and virtual address spaces, virtual address cannot be mapped directly to locations in a certain guest program, but instead this translation depends on the current state of the guest system. To represent this relation it is necessary to adopt the introspection to the current (introspection) context of the guest.

Chapter 4

Design

This chapter covers the design and motivation of our approach. As we already stated in our analysis (§ 3), we aim to build an event based introspection approach which uses debugging symbols to bridge the semantic gap with only a small impact on the simulation performance. To achieve high performance, all debugging symbol information is parsed off-line and the derived instructions and breakpoints are converted to machine code. We represent the changing context within the guest machine by sets of *Introspection Modules (I-modules)*. Each I-module is responsible for a single program, library, kernel module or the kernel. An I-module is loaded into the current set when the code and data of the corresponding program is currently represented in the virtual address space used by the CPU.

We begin this chapter with a description of the required input files in Section 4.1. The subsequent Section 4.2 depicts the components relevant for building I-modules based on the input. This chapter ends with Section 4.3 discussing the interface between the I-modules and a full system simulator.

The overall workflow from input files to running of introspection code is depicted in Figure 4.1. The details on the different stages are described in the individual sections.

4.1 Input and Required Files

Three different sources of input are required for the process of creating I-modules. The first is the debugging information for the program or kernel we want to instrument. As the instructions necessary to access data within the guest are pre-computed, the accessed variables and the required breakpoints must be defined in advance. They are specified in a second input file, the *Runtime Position Configuration* (§ 4.1.1). The third input source is the *User Defined Code* (§ 4.1.2) representing the actions the creator of an I-module wants to execute on different events, e.g., accessing a variable previously defined in the configuration file and recording its value.

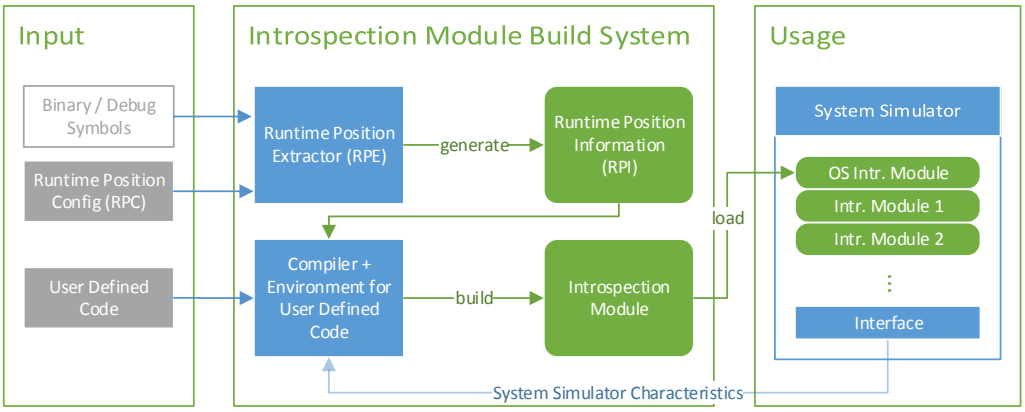

Figure 4.1: Basic workflow from user provided input files to the loading of a compiled I-module.

4.1.1 Runtime Position Configuration

A Runtime Position Configuration (RPC) file describes the locations where we want to insert breakpoints and which variables we want to be able to access. In addition, it is possible to name structures within the executable's source code we want to be able to access and traverse, comparable to traditional snapshot based introspection approaches (§ 2.2.1). Every entry in a RPC file has a user defined symbolic name and this entry, e.g., a breakpoint or a variable at a breakpoint, is referred to with this name in the User Defined Code.

An arbitrary number of breakpoint locations can be specified and for every location, several variables then accessible in the User Defined Code can be named. A variable is not necessarily a variable name in the current context, but it can be an *Access Path* starting with such a variable name as well.
These paths can contain instructions to resolve pointers and to select members of structures. An example for such a path is "`task_array[3]=>pid*`". This path selects the 4th element (in this case a `struct`) within an array called `task_array` and accesses its member named `pid`. The asterisk states, that we want to access the value of this variable, not its address.
To allow more flexible description of such paths, it is possible to replace array indices with variables, which can be specified when accessing these paths in the User Defined Code.

As the Access Paths are specific to an exact code location, they can take care of problems like values held within a register at a certain location in the program. However, they are limited to paths known in advance. It is for example impossible to write an Access Path which allows traversing all `task_structs` within the Linux kernel. The number of pointers followed within an Access Path is user defined, but fixed. Hence it is not possible to traverse a linked list with an unknown number of entries.

To enable such traversing of linked structures and their members, the author of an I-module wants to access, they can be specified in the RPC file. The offsets and types of the structures' members are then usable within the User Defined Code by symbolic names to walk linked structures manually.

4.1.2 User Defined Code

While the RPC file specifies what we want to access and where we want to set breakpoints, the User Defined Code describes what to do on their occurrence. This code is normal C code with additional functions enabling for example high-level access to variables or direct access to the guest's memory.
The individual breakpoints are represented by a particular function called at each occurrence of the breakpoint. Within such a function we provide certain information on the current state of the system. The detailed information available to the user depends on the full system simulator used and its interface. However, there are some standard variables the environment provides, e.g., the identifier of the CPU hitting a breakpoint. The User Defined Code should normally not need to access the non-standard information directly, and hence this does not necessarily result in full system simulator dependent code. However, the implementation of the functions provided by the environment may depend on the full system simulator used. This allows to implement simulator design specific features, e.g., accessing the value within a guest register without the need for a function call into the simulator interface.
Within a function representing a breakpoint, it is possible to specify types of variables by using the symbolic names specified in the RPC file. The current values of these variables can also be extracted automatically by using their symbolic names. Figure 4.2 contains an example.

```
1 BREAKPOINT_HANDLER_START(my_bp_name)
2   GUEST_TYPE(my_var_name) nr = GET_GUEST_VARIABLE(my_var_name);
3   printf("cpu: %d, number in guest: %d\n", cpuNr, (int) nr);
4 BREAKPOINT_HANDLER_END()
```

Figure 4.2: Example on how a User Defined Code part representing a breakpoint could look like. Uppercase expressions are not plain C code, instead they are automatically replaced by C code during the build process of an I-module. The `my_*` parameters are the symbolic names used in the RPC file.

4.2 Introspection Module Build System

The introspection module build system consists of two components together allowing the translation from the input files into *I-modules* (§ 4.2.1). The first component is the *Runtime Position Extractor* (§ 4.2.2) which analyzes the debugging symbols and extracts the required information. The second is the *Environment for User Defined Code* (§ 4.2.3)

necessary to assemble the extracted information and the provided code to a runnable module. Figure 4.3 depicts the overall build process.

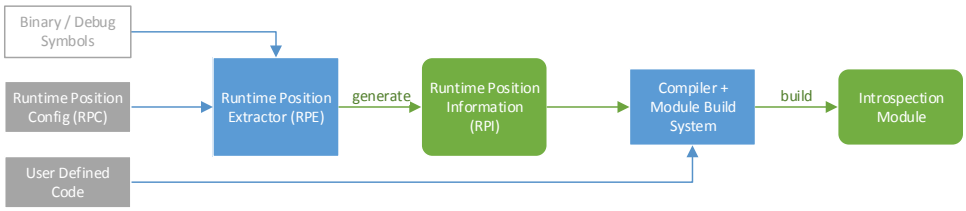

Figure 4.3: Stages and Components of the build process. The debugging symbols are parsed according to the RPC file; the extracted information is saved to a RPI file. This file provides the necessary knowledge for the Module Build System to compile the User Defined Code into I-modules.

The module build system is not run normally during the introspection but the I-modules are created in advance instead. Still, it is possible generally to load newly created I-modules into the running simulator.

4.2.1 Introspection Modules

An I-module encapsulates the data required for introspecting a program, library, kernel or a kernel module. This includes the necessary breakpoint locations and the instructions to execute on such an event. In addition, an I-module also includes a function that can be called to check whether a module is relevant for a certain guest system context.

Pre-calculated addresses (e.g., in a library) are problematic as they are not necessarily known at compile time, but are chosen on load time instead. Normally a loader moves all addresses of a binary relative to an offset [34], but this offset may be different for several instances. Reasons for different offsets can be a possible collision of libraries with other binaries or the usage of *address space layout randomization*. See Figure 4.4 for an example.

To cope with this issue we do not use I-modules directly, but we use *instances of I-modules* instead within the individual guest contexts. As depicted in Figure 4.5, the instances contain the relocation information for the I-modules.

4.2.2 Runtime Position Extractor

Runtime Position Extractor (RPE) is the component responsible for the extraction of relevant information from the debugging symbols. Its inputs are the RPC file and the debugging symbols. The extracted information is saved into a RPI file.

It contains addresses for breakpoint locations, instructions on how to access values of certain variables at these breakpoints, type information on these variables and information on type and offsets for structure members.

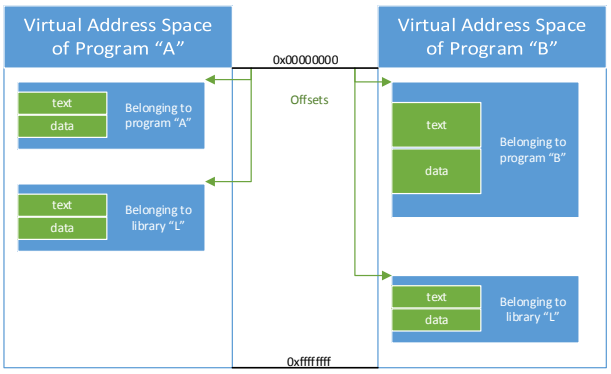

Figure 4.4: When a loader loads code into an address space, it does not necessarily place it on the location the linker chose. To avoid overlapping, e.g., when loading libraries in Linux, a library or program can be loaded at different addresses in several address spaces.

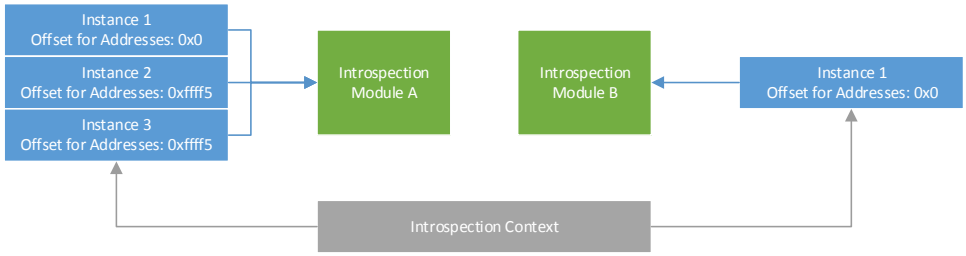

Figure 4.5: Programs currently introspected are represented by a set of I-module instances. The I-module instances contain information on single instances, for example of a program. The minimal required information is the offset for all addresses used in the I-module.

Runtime Position Information

While a RPC file is meant to be written by hand, a RPI file contains the low-level information generated automatically, necessary to implement the actions described in the RPC file. This includes breakpoint addresses, instructions on how to access variables and type information for variables. Type information refers to characteristics like byte size, encoding and endianness. There is an important difference between these two representations. A breakpoint location in a RPC file can result in several breakpoint addresses in a RPI file. This can be caused by a breakpoint set in a function inlined at several locations. Figure 4.6 depicts an example for the relation between both files. In the following we will refer to breakpoints which describe locations within the source code as *logical breakpoints*.

In addition to this breakpoint centered data, the RPI file also holds the offsets within the structures described in the RPC file.

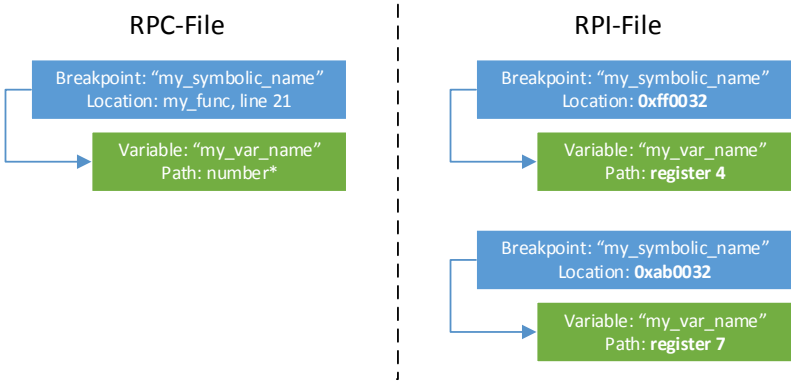

Figure 4.6: A logical breakpoint location specified in the RPC file can be represented by several breakpoint addresses in a RPI file. The instructions how to access the named variables depend on the individual breakpoint address and hence are also stated several times.

4.2.3 Environment for User Defined Code

In order to compile the User Defined Code, the callback functions for breakpoints and the functions allowing to access guest variable content by symbolic names must be provided. The representation of breakpoints must in particular be able to cope with problems arising due to possible interruptions at the instruction.

Breakpoint Representation

In general it is sufficient to represent a breakpoint by a single function within the code. This function is called on a hit of one of the addresses representing the breakpoint. However, a problem arises when we take into account that an instruction might become interrupted, e.g., because of a page fault.

Interrupted Instructions A breakpoint set at a specific address results in calling a debugger *before* the instruction at this address has actually been executed (see Figure 4.7) [29]. Every time the CPU tries to execute the instruction, first the debugger is called and the execution of the instruction is postponed until the debugger finishes its work.
Assuming the instruction causes an interruption, the instruction might be restarted, e.g., in case of a valid page fault. If the instruction is restarted, the debugger would be called again, too. Calling the debugger twice is misleading as this instruction is only executed once from the perspective of the program flow.
Normal debuggers can solve this by using a special flag of the CPU, e.g., the *resume flag* on the X86 ISA [27], which prevents them from being called twice in case of an interruption. However, this is not a solution in our case, as we do not want to affect the guest CPU or rely on state the guest system can modify. In addition such a solution would be ISA specific. Introducing a shadow flag is also problematic as the original flag used by

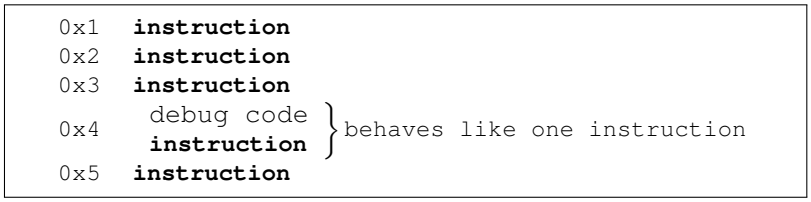

Figure 4.7: Location of a breakpoint. When the instruction at 0x4 is interrupted, it *might* be restarted in case of a page fault, but the execution can also continue at another location in cases such as a division by zero. However the location where the execution continues depends on the OS's decision.

the CPU is saved on the stack in case of context switches. Duplicating such a functionality without modifying the guest is very difficult.

Passing Additional Information after the Instruction's Execution The decision whether the CPU restarts the execution of a program after an interruption at the same instruction or at a different location, e.g., at an exception handler, depends on the OS. As a result it is not easy to tell in advance whether an interrupted instruction will be executed a second time or if the execution will continue at a different location. Also when the breakpoint handler for an instruction is called, there is no general way to know whether the breakpoint handler function was semantically executed already.

We are therefore prone to calling the breakpoint handler functions several times when it should be called logically only once. This may lead to improper results if the number of issued breakpoints itself is an important part of the state of the guest. For instance, this number may be relevant in cases where the number of executions of a function is measured by inserting a breakpoint at its beginning.

To cope with this issue we introduce a second callback function for each breakpoint which is called after the breakpointed instruction was executed. This function has a parameter encoding the information whether the instruction was normally executed or if it was interrupted. This information is available on a hardware level and thus we can easily gain access to it.

The user can now define the appropriate reaction, e.g., the information collected on a breakpoint can be discarded, if the instruction itself was interrupted. This decision is not correct in general. If the introspected code uses an exception handler to implement software constructs like *software* exceptions, the instruction might not be restarted but instead execution might be continued at the software exception handler and hence the breakpoint event would be ignored completely.

However, this is correct when the relevant code for the introspection does not use exception handlers intensionally. If the situation is more difficult, it might be necessary to add breakpoints to the software exception handlers to be able to decide what to do with a breakpoint event.

In case only the variables' values are important and the number of breakpoint hits does not matter, it is a valid option to record the results of a breakpoint regardless of the information provided by the second callback function.

High-Level Interface: Symbolic Access

The main task within a breakpoint handler function in the User Defined Code is to inspect the state of the guest. While we provide direct access to the guest state, the user code can implement the data extraction on a much higher level.
For every breakpoint location several variables can be specified in the RPC file and their type information and data (automatically extracted from the guest's state) can be used within the individual function representing the breakpoint. The module system obtains the necessary information to provide this functionality from the previously generated RPI file.
Within every handler function certain important context information is available. The most important is the current guest CPU identifier on which the breakpoint occurred. It allows associating the calls to the first and the second breakpoint handler, even if several CPUs are simulated in parallel.

4.3 Full System Simulator Interface

The *Full System Simulator Interface* connects the individual I-modules with the full system simulator. The build system provides high-level interfaces for the User Defined Code, but requires low-level access to the guest system itself, in order to read the guest memory at a certain location or to access register values of the guest. In addition, the Full System Simulator Interface must provide a way to decide which instances of which I-modules should be currently used and a method to find I-modules relevant in a certain context.
In the first part of this section we look into the representation of a context. In the following we describe the Low-Level Interface for access to the guest state and the selection of I-module instances. This section ends with considerations on the implementation of breakpoints. Figure 4.8 depicts the interaction between the interface and individual I-modules and hence represents all relations described in this section.

Context Representation

A handle for the current context is passed when we call functions representing the breakpoints within an I-module and when the I-module calls functions of the Full System Simulator Interface. The context consists of:

1. Information on the machine state (e.g., the CPU identifier or register values).

2. Information on the I-module instances currently in use.

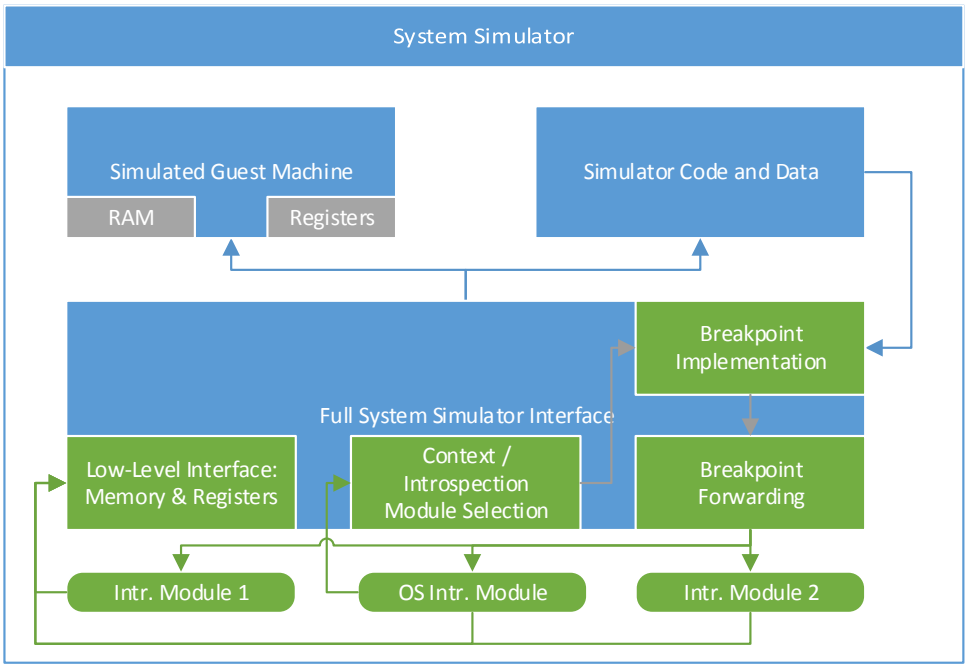

Figure 4.8: Communication between I-modules and the full system simulator by using the Full System Simulator Interface. Every I-module can use the interface for specifying the set of current I-module instances, but as this task is intended to be done by the OS I-module, the access is only depicted for this I-module.

4.3.1 Low-Level Interface: Memory & Register Access

The *Low-Level Interface* allows accessing the guest state, e.g., the memory content or register values. The RAM is accessed by its *virtual* address as it is done within a debugger. This means the memory accesses as well as register accesses depend on the state of the guest machine, i.e., the state of the CPU in whose context we want to access the memory. When accessing virtual addresses the translation is done by the interface, this implies that an access might fail because there might be no valid translation for a certain virtual address. Therefore all low-level memory accesses return whether the requested operation has succeeded.

4.3.2 Introspection Module Selection

As mentioned in the analysis chapter the selection of an introspection context (the set of I-module instances) is done by introspecting the OS kernel. This introspection is done by providing an OS I-module which is basically a normal I-module responsible for the OS kernel and which is part of every introspection context. We use the term OS I-module because this module has a special semantic role within the design, but its implementation is similar to other I-modules.
To select the I-modules for a certain context, it is necessary to be able to list and commu-

nicate with all other I-modules. The OS I-module (or any other I-module) can query all other I-modules and check if an instance of them should be part of the current context. On the creation of a new process for example, the OS I-module can search whether there is an I-module built to introspect this specific process. How a process is identified is an agreement between the I-modules and is not part of the overall design, but in general the program name and path might be a sufficient identifier.
To add an I-module to an introspection context, an instance must be created and attached to the context afterwards. The Full System Simulator Interface must provide functions for both.

4.3.3 Breakpoint Forwarding

Depending on the current context the Full System Simulator Interface must interrupt the execution of the guest machine when hitting a breakpoint and call the appropriate function within one of the I-modules. This implies that the implementation of breakpoints is located in the Full System Simulator Interface or it uses some functionality of the simulator to provide this feature.
To implement (instruction) breakpoints it is necessary to check every IP value before the actual execution of the instruction at this address. This check can impose a huge performance impact and thus its implementation must be optimized for performance.

4.4 Conclusion

Debugging symbols provide detailed information on the machine code and allow to bridge the semantic gap even for event-based introspection. The design of our introspection approach consists of two parts. The first is the Introspection Module Build System which builds I-modules based on configuration files and code provided by the user. The configuration files specify breakpoint locations and the guest variables used within the User Defined Code. In this code, breakpoint handlers can be specified and the described variables can be accessed using their symbolic names. The debugging symbols, used to determine the actions necessary to extract guest variable values, are parsed during the build process and transformed into machine code. Thus the time consuming analysis is done off-line and not during the introspection.
The second part is the Full System Simulator Interface providing an environment for loading and using the I-modules to introspect a guest system running inside the simulator. Besides allowing access to the guest state, the interface is also responsible for switching between different sets of I-modules which represent the current introspection context. While the speed of the implementation of the build system is not important as it is run off-line, the implementation of the interface must focus on minimizing the impact on the simulator, as it is executed within the simulator's context and so deteriorates the simulation performance.

Chapter 5
Implementation

This chapter contains the description of our implementation according to the design discussed in Chapter 4. Though the implementation is extensible to different host, guest and simulator systems, the current implementation focuses on a Linux target using QEMU. Details on the environment are discussed in Section 5.1.

Whenever possible we divide the implementation into a platform independent and a platform dependent part. The components to be covered are in particular the Runtime Position Extractor (RPE) (§ 5.2 and § 5.3), the Introspection Module Build System (§ 5.4 and § 5.5) and the Full System Simulator Interface (§ 5.6 and § 5.7).

5.1 Target Environment

The host and guest ISA for which we implement the ISA dependent parts is AMD64. Linux is a popular OS and the DWARF debugging symbol file format used is well documented. Hence starting with supporting this combination seems reasonable. We use Linux as the host OS which executes the simulator.

As we previously identified the performance of the introspection to be a critical feature, we chose QEMU as a full system simulator. It is very fast and in consequence the slowdown factor caused by an introspection approach is likely to be near to the worst case for all simulators.

However, as the guest simulated by QEMU is influenced by all software running on the host (§ 2.1.3), it is not possible to implement an introspection approach for this simulator that completely avoids modifying the timings of the execution within the guest. This is not a limitation of our implementation, it can be solved by using a timing-accurate simulator.

5.2 Runtime Position Extractor Implementation: General

The task of the RPE is to extract debugging symbol information for items, named in a Runtime Position Configuration (RPC) file, and to save the required details into a Runtime Position Information (RPI) file. While the syntax of the RPC and RPI file does not depend on the type of debugging symbols, the code which analyzes the symbols is written for the individual symbol format.
To support several debugging symbol file formats, we provide the debugging symbol parsers as plug-ins (§ 5.2.2) employing a representation of a RPC file (§ 5.2.1). This representation is called *Expression Container* and once a plug-in has enriched it with the extracted information, it can be converted to a RPI file (§ 5.2.3). Figure 5.1 depicts the high level interactions between the various components.

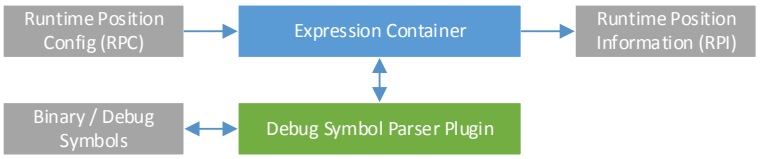

Figure 5.1: Components of the RPE implementation. To avoid unnecessary duplication of code, the individual implementations of debugging symbol parsers are isolated from the reading and writing of RPC and RPI files.

As RPE is not meant to run during the simulation, the performance of its implementation is of secondary importance. We therefore trade speed for a more general solution.

5.2.1 Runtime Position Configuration

The RPC is a simple configuration file which we read using *libconfig* [35]. The data contained in this file such as breakpoint locations and structures, is already defined by the design. Only the syntax of *Access Paths* for variables is specific to the implementation and hence we describe it in the following section.
An example for a complete RPC file can be found in the Appendix (Figure A.1, p. 86).

Access Paths

An important part of the configuration file is to describe which variables should be accessible for the User Defined Code. A symbolic name is attached to each Access Path to allow accessing it within the User Defined Code.
As mentioned in the design chapter, an access path must allow to encode a path from a variable found in the current context to the data we want to access. We encode this in a format using a syntax similar to normal C code. However, there are differences. Among others we use a single symbol to access the members of a structure regardless of the current object being a pointer or the actual structure. The general syntax of an Access Path is defined as:

variable-name-in-code[=>*entry-name*|=>|[*X*]]*[*|]

The bold blue symbols are not part of the syntax, but are used instead to describe the different possibilities. We describe the symbols which are part of the Location Description in Table 5.1.

Symbol	Description
no modifier	At end of the Access Path: Get address of (virtual) guest memory where this object starts. This may fail, as the object may reside in a register and hence it may not have an address. Example: Object is "`int a;`" — "`a`" describes the guest address where the value of "`a`" is located.
`*`	At end of the Access Path: Get object. Example 1: Object is "`int a;`" — "`a*`" describes the value of "`a`". Example 2: Object is "`int *a;`" — "`a*`" describes the value of "`a`" that is the value of *a pointer* to value "`a`".
`[X]`	Array entry selector, if empty the value must be passed at runtime.
`=>`	*First meaning:* Follow pointer. Example: Object is "`int *a;`" — "`a=>*`" describes the value of the integer "`a`" points to. *Second meaning:* If a modified object is a structure or a class, "=>" is discarded, i.e., it is only used to separate two strings. Example 1: Object is "`struct str_f { int a; } foo;`" — "`foo=>a*`" describes the value of integer "`a`" contained in the structure. Example 2: Object is "`struct str_f *fb;`" — "`fb=>a*`" also describes the value of integer "`a`".

Table 5.1: An overview on symbols within an Access Path. Syntax for casting is omitted.

5.2.2 Plug-in System

The debugging symbol parser plug-ins use an Expression Container to obtain instructions and to export the extracted information. We depict an example for the data contained within such a container in Figure 5.2.
Items not represented at all within a RPC file are the *Relocation Descriptors*. They describe addresses and their IDs can be used instead of real addresses within other expressions. As a benefit we are able to encode additional information, for example in which logical segment they are located. This information is not required normally, as all segments are moved together by a single offset and hence it is irrelevant in which logical segment the addresses reside. However, the loader for Linux Kernel Modules (K-modules) moves the single logical segments relative to each other and so we need this information in this special case.

The remaining entries such as *Location Expressions* and *Structure Expressions* are a representation of breakpoints and structures described in the RPC file.

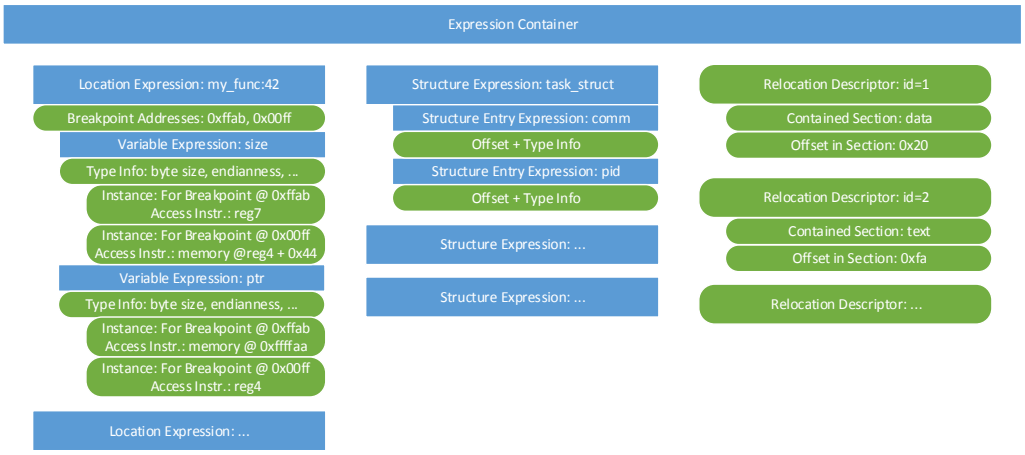

Figure 5.2: A simplified representation of the data contained in an Expression Container. The blue rectangles describe information contained in the RPC file, the green ovals represent data added by the debugging symbol parsers.

5.2.3 Runtime Position Information

Most of the information we want to represent in the RPI file is simple and the range of possible values is obvious, e.g. the size of a variable is an unsigned number and the endianness of a variable can be either little or big.

This is not true for the description of actions necessary to access the value of a variable. In theory these actions can be arbitrarily complex. The DWARF format for example encodes this information as a program for a stack machine (§ 2.3.1). To be able to represent complex dynamic instructions we also encode this information by a program consisting of self-defined instructions, we call them *Location Description Instructions*. These instructions contain support for a stack, access to the guest's memory and registers and several calculation operations. Though the defined instructions were sufficient for our purpose, the supplementing of new instructions is a possible part of supporting further debugging symbol formats.

We provide an example for a RPI file in the Appendix (Figure A.2, p. 87). It is noteworthy that the format of this file encodes the contained information redundantly. This is necessary, as using this file within the Module Build System would be difficult otherwise.

5.3 Runtime Position Extractor Implementation: DWARF Specific

The DWARF specific implementation is a plug-in for the symbol parser. Because of the huge scope of this format, currently we only provide partial support for DWARF. We implemented only the parts of the standard which where necessary during our tests and measurements. For example we do not support DWARF expressions which jump to other DWARF expressions as this never occurred within the debugging symbols we used.
We parse the binary representation of the DWARF format utilizing *libdwarf* [24] which presents the contained information as C/C++ structures.

5.3.1 Breakpoint Address Localization

The first task of the parser is to translate logical breakpoint locations to actual breakpoint addresses. DWARF provides the mapping from positions within source files to addresses via the line number information (§ 2.3.1). Generally it is sufficient to specify a breakpoint location by the file name and the line number. We additionally require the name of the function containing the breakpoint as this simplifies the search within the debugging symbols.
We start with the search for the Debugging Information Entry (DIE) representing the function containing the breakpoint. If this function has several code instances, e.g., because it is inlined, we also search for the DIEs which represent the individual instances. Every DIE resulting in executable code contains information on the (not necessarily continuous) address ranges populated by the instructions of the function's instance. We search the line number information of these ranges for the addresses representing the logical breakpoint location.
After this has been completed the actual breakpoint addresses representing each logical breakpoint are known.

5.3.2 Variable Support

While the variables' type information only depends on the logical breakpoint, the instructions on how to access their values may be different for each breakpoint address representing a single logical breakpoint.
As the DIEs for the different function instances have already been located, we can now search in each of their subtrees for the DIEs representing the variables requested by the RPC file. This results in a DIE for each breakpoint address and variable. Starting from these DIEs the Access Paths described in the RPC are evaluated and the DWARF expressions describing how to follow the paths are memorized. Even if the DIE for a variable is the same for each function instance, the resulting DWARF expressions can differ as a DIE can contain a list of such expressions. Which expression is to be used depends on the IP (which is the breakpoint address in our case and which differs for each function instance).
DWARF describes locations where values will be located during runtime by a program evaluated on a stack machine. A simple location description is not sufficient in general,

for example a local variable is likely to be found at a memory location relative to the frame pointer and therefore describing this location with a hard-coded address is impossible.
In order to have a debugging symbol independent encoding of this information, the memorized DWARF expressions are transformed to our self-defined Location Description Instructions. This transformation is easy as we provide similar instructions like DWARF does.

Additional to the information on how to access a variable, it is also necessary to know its type information such as the size or the encoding. The evaluation of each Access Path traverses the DIE tree starting from a DIE of a variable or a parameter. The DIE at which we reside when the complete path is processed represents the variable whose value the user wants to access and hence this DIE contains the type information of the variable described by the Access Path.

5.3.3 Structure Information Extraction

As mentioned in the design chapter, Access Paths cannot be used to traverse structures such as linked lists with an unknown number of entries. However, linked structures can be walked by manually following the pointers between them in the guest memory. This requires information on the layout of the structures and their member variables. DWARF holds the required information on every structure, used within the program code, in the DIE tree. The structures' children describe the offsets and names of the members. Additionally, they point to DIEs which contain type information for the members.
The necessary information is collected for the entities named in the Expression Container and the extracted details are attached to the Structure Expressions.

5.4 Introspection Module Build System Implementation: General

Once the RPE has analyzed the debugging symbols and has used the relevant parts to build the RPI file, all files necessary to create an Introspection Module (I-module) are available.
We implement the I-modules as shared libraries which we build using GNU Compiler Collection (GCC). The environment providing the symbolic access functions consists only of header files and preprocessor macros which generate C functions and structures based on the content of the RPI file. The functions and the code we generate is ordinary C code and it is compiled with the rest of the module.

This section starts with describing how an I-module can signalize its responsibility for a certain introspection context (§ 5.4.1). Afterwards we discuss the implementation of the breakpoint handlers in the User Defined Code (§ 5.4.2) and the realization of macros providing access to guest variables (§ 5.4.3). The section closes with a short note on walking of guest structures (§ 5.4.4).

5.4.1 Introspection Context Membership Decision

As described in the design, each I-module can be asked by the OS I-module whether it considers itself part of an introspection context. To implement this interface, each I-module must provide a callback function named `vp_check_match(...)`. The OS I-module can call this function of all other I-modules through the Full System Simulator Interface. The parameter of this function is a pointer referring to a structure describing the current introspection context. This structure is defined by the implementation of the OS I-module and hence all other I-modules must share a header file with the OS I-module to access the contained information. An example for the information passed could be the name of the binary or library started within the introspection context.
Whether an I-module wants to be part of the context is indicated by the returned value.

The I-modules can provide additional callback functions called on events such as the initial loading of the I-module or the creation of an instance. Most of these callbacks are optional and the Full System Simulator Interface detects automatically if they are defined or not.

5.4.2 Breakpoint Environment

The environment for logical breakpoints can be looked at from two perspectives. One is the viewpoint of the User Defined Code. We included a snippet of this code in Figure 5.3. The available information and functions are simulator and platform independent.

```
1  #define BP_HANDLER_NAME task_create
2  GET_TYPE(pid)    *procID; // initialized elsewhere
3  BP_HANDLER_START
4     int flags = 0;
5     procID[cpuNr] = GET_DATA(pid, &flags);
6     [...]
7  BP_HANDLER_END
8
9  BP_SECOND_HANDLER_START
10    if(INST_WAS_FULLY_EXECUTED)
11          [...]
12    else
13          [...]
14 BP_SECOND_HANDLER_END
15 #undef BP_HANDLER_NAME
```

Figure 5.3: The layout of a breakpoint handler in the User Defined Code. This is a handler for the logical breakpoint named “task_create” and an access to a guest variable is depicted. The second handler which is informed whether the instruction after the breakpoint was successfully executed starts in line 9.

The other perspective on this environment opens after the evaluation of the `BP_*` macros. These macros generate two functions which are called by the Full System Simulator

Interface on the occurrence of the logical breakpoint. The generated functions initialize structures and variables describing the guest's CPU and its state such as the `cpuNr`.
All hidden information and structures are encapsulated in a C struct called `bp_env_t` and a pointer to it is passed transparently when using a macro such as `GET_DATA(...)`.

In addition, a buffer is created for recording pointer addresses describing memory allocated without the knowledge of the user. This is necessary as the macros which access the guest (such as `GET_DATA(...)`) sometimes pass and receive buffers when communicating with the Full System Simulator Interface. As these buffers were not requested in the unexpanded User Defined Code, they cannot be freed manually. Therefore they are collected and the addresses are freed by code generated by the `*_HANDLER_END` macros.

5.4.3 Accessing Symbolic Variables at Breakpoints

In a breakpoint handler function it is necessary to be able to create variables holding the data which will be read from the guest system. To support that, we provide several macros such as `GET_TYPE(sym_name)` or `GET_SIZE(sym_name)`. They expand to a type or to the byte size of the variable with the name "`sym_name`" defined for the current breakpoint. These macros can be used to write code without hard-coding the detailed representation of guest variables, such as the byte size. This can be useful when the same User Defined Code shall be compiled for different guest ISAs, as for example the byte size of a variable used within a program may change depending on the ISA.

Structures for Easy Guest Variable Access

A single logical breakpoint can result in multiple breakpoint addresses (§ 4.2.2). To allow the User Defined Code to hide the existence of multiple breakpoint addresses, each breakpoint belonging to the same logical breakpoint is delivered to the same breakpoint handler. These different sources for a breakpoint are represented by a hidden breakpoint ID which is passed to the breakpoint handler function.
As the actions necessary to access the value of a guest variable depend on the breakpoint address, every access to a guest variable must be redirected to breakpoint ID dependent instructions.

We provide different macros to access each guest variable. For example, the `GET_DATA(sym_name)` macro returns the value directly, whereas the `GET_DATA_PTR(sym_name)` macro returns a pointer to memory containing the guest variable or structure. Further versions of these macros accepting parameters are necessary, to allow passing parameters to the Access Path. This is required to traverse arrays (§ 4.1.1). However, all macros accessing the same variable rely on the same debugging symbols for the same breakpoint ID.

To represent this relation between the different macros, the breakpoint ID and the Location Description Instructions, the macros expand to a cascade of function calls. Figure 5.4 depicts the resulting structure.

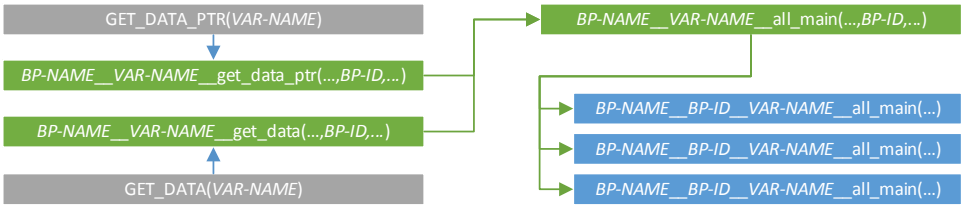

Figure 5.4: The function cascade hidden by macros for accessing guest data at breakpoint locations. The macros expand to function calls to macro and context specific functions. These functions use a common helper function which calls another function depending on the breakpoint ID in order to extract the data from the guest.

The name of the functions, which are called by the macros, contains the name of the logical breakpoint and the name of the variable we want to access. Most information like the breakpoint name and the breakpoint ID used as a parameter is not provided by the parameters of the macro but is extracted from the context. While the breakpoint name is known from the position where the macro is used, the breakpoint ID is a variable provided in the function representing the breakpoint environment.
The function which is implicitly called by the macro does not access the guest data directly, but it only takes care of passing the data once it is acquired instead. All functions representing a macro for the same variable call the same function to access the guest.
This shared function passes the task of extracting the guest's variable content to another function. Which function it choses depends on the breakpoint ID. The ID allows to differentiate between different breakpoint addresses for the same logical breakpoint. The breakpoint ID specific functions finally contain the instructions from the RPI file describing how to access the data. These instructions use the Full System Simulator Interface to access data of the guest such as the memory content at a certain location.

All functions called in this cascade are generated based on the RPI file using the macro preprocessor. As there are several function calls in that process and most are only used once, we ask the compiler to inline them.

Translation of Location Description Instructions

The breakpoint ID specific functions mentioned above need to execute our self-defined Location Description Instructions introduced in Section 5.2.3.
Instead of providing an interpreter for these instructions, we wrote C macros expanding to C code which implements the individual functionality. For example a Location Description Instruction accessing the guest memory is expanded to a call to the Full System Simulator Interface. Simpler instructions like add or shift are converted to equivalents in C.

As the Location Description Instructions are implemented as macros, it is possible to copy them into a function and thus they are automatically translated to valid C instructions before the source code is compiled. The Location Description Instructions generated by RPE are not optimized and hence they may contain unnecessary instructions or instructions which could be simplified. For example several subsequent additions of hard-coded numbers could be represented by a single addition; "`PUSH(ADD(POP, ADD(21, 21)))`" can be expressed as "`PUSH(ADD(POP, 42))`".
As we do not interpret the instructions, but the generated C code is compiled instead, the optimization of the compiler is used indirectly to optimize the Location Description Instructions.

5.4.4 Walking of Structures

We currently provide only little support for walking memory structures as this functionality was not needed in our scenarios and the problem itself has already been solved by other introspection approaches (§ 2.2). However, the module environment contains macros to obtain offsets of structure members and to read the guest memory. This is sufficient to walk guest structures, but a more user friendly interface would be appropriate for every day use.

5.5 Introspection Module Build System Implementation: QEMU Specific

The QEMU specific part of the build system is rather small. The only parts depending on QEMU are the data types used in the Full System Simulator Interface, and the implementation of guest register access.

5.5.1 Interface Data Types

To communicate with QEMU we must use common data types, e.g., to represent guest and host addresses. Our build system implements the data types for this exactly as done by the specific full system simulator. We do this by including parts of QEMU's headers. The module system represents address types by macros and type definitions internally. Hence we are able to change them to types and sizes used by the individual simulator — in this case QEMU.

5.5.2 Register Access

It would be possible to use an interface function to access the state of a certain register. However, this would result in an unnecessary function call as in QEMU's case the CPU state and hence the registers are accessible from within the module system. We therefore implement the register access by a QEMU specific macro which extracts the value of the register directly from the pointer to the current CPU state.

Registers may prove problematic as they may be numbered differently in the debugging file format and the simulator. Hence a translation might be necessary. This problem also arose with QEMU.

5.6 Full System Simulator Interface Implementation: General

The interface implementation must be focused on speed as it is used during the runtime of the simulator — in contrast to other components like the RPE. The most time critical part of the interface is the check for possible breakpoints. To minimize the time spent for these checks we implemented a hash-map based approach (§ 5.6.1). Because of pre-computations necessary for this approach, it is not reasonable to rebuild the set of I-modules on every context switch. We discuss this problem and its solution in Section 5.6.2.

5.6.1 Fast Breakpoint Check

Every guest instruction executed by the simulator must be checked against all breakpoints relevant for the current context. Implementing this by comparing the current IP with all breakpoints of all I-module instances would result in traversing a lot of structures. In addition, the amount of time necessary would increase linearly with the number of breakpoints ($O(\#breakpoints)$).

The effort for the lookup can be reduced by using a hash table with the address as the key and pointers as the values, referring to arrays of breakpoints with the same key. Conversion of addresses to keys is discussed in the following section.
Assuming the map is large compared to the amount of breakpoints, most checks for a breakpoint hit can be rejected solely by a lookup in the hash-map and hence they can be done in linear time. Additionally, it is not necessary to search all breakpoints on a hit, but only the breakpoints mapping to the hash table entry found must be searched. However, in case of a hit, the work necessary for checking breakpoints is still $O(\#breakpoints)$ in the worst case, i.e., if all breakpoints map to the same key. But this is rather unlikely and we still would improve the performance by directly rejecting a lot of addresses before this check.

We want most values of the hash table to be empty, as this allows to reject addresses most rapidly. However, if most table entries were empty, a lot of memory would be wasted. As every entry of the hash table is a pointer, the memory for a complete pointer is used to encode that there is no breakpoint (i.e., by containing a null-pointer). To reduce the memory consumption without reducing the number of directly rejected addresses, we generate two different tables. One to achieve a high reject rate and one to reduce the number of breakpoints to search on a hit.
The first table only indicates whether a hit occurred. As this information can be encoded using a single bit, the number of entries can be large while still resulting in a moderate

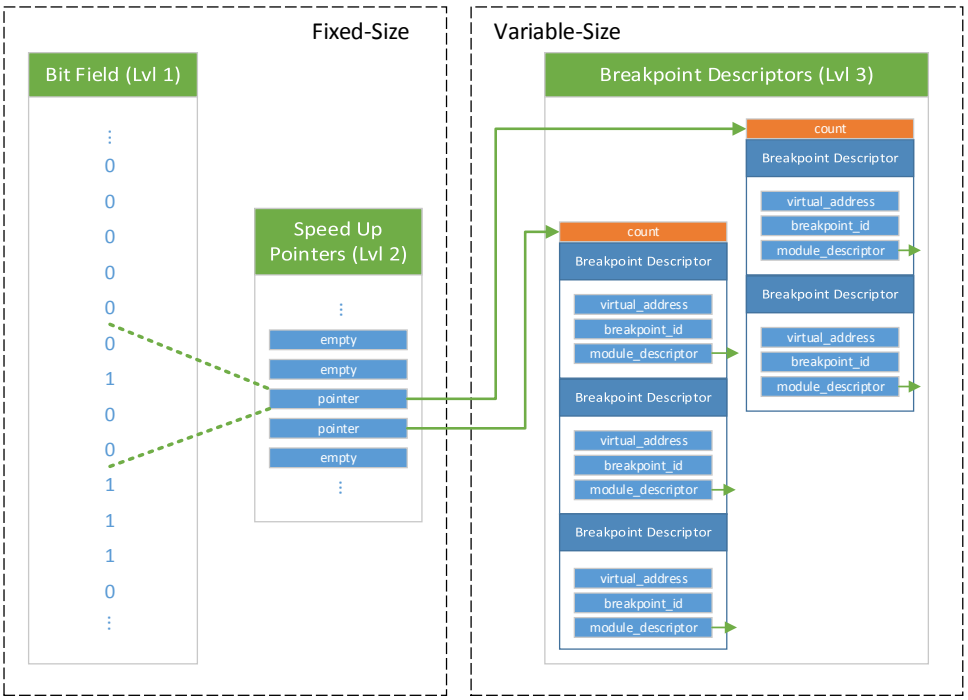

Figure 5.5: Cascade for checking whether an address is marked to be a breakpoint. Advantages: (i) A lot of addresses can be rejected solely by using the first level table. (ii) Often the breakpoints should map to different second level entries, in this case we must only compare a subset of the breakpoints against the tested address.

memory consumption. In our current implementation, the first level table has 2^{15} entries resulting in a memory consumption of 4 KiB. However, the size of the tables can be configured at compile time.

The entries in the second table are pointers and hence they can refer to arrays of breakpoints. As the size of a single table entry is much larger, we only use 2^{10} entries per table. The resulting memory consumption is 8 KiB when using 64 bit pointers.

In our case, 32 entries of the first level table map to the same entry in the second level table. The layout of the used structures is depicted in Figure 5.5.

To check for a breakpoint, the key for the first table is calculated and the indicated bit is checked in the first level table. In case of a miss, the address is rejected. In case of a hit, the key of the second table is calculated and the pointer belonging to this key is used to find the array of breakpoints. Each entry of this array is then checked against the current IP.

Key Derivation from Addresses

The keys for both hash tables are a part of the breakpoint address. The number of entries of the hash tables is a power of two and hence the key can be calculated solely by shift operations.
We divide an address into 3 parts: `0xI|J|K`. The upper bits (`I`) should not be used as the key, because all addresses within a program could map to the same table entry otherwise (in particular if the address size is 64 bit). For ISAs where the instruction length is fixed and the instructions must be aligned to the instruction size, some of the lowest address bits are equal for all breakpoints. Using these bits as part of the key would not be reasonable as some entries of the hash table would never be used in this case. Therefore some lower bits (`K`) are also removed.
The remaining part of the address (`J`) is used as the key for the hash tables. The size of `J` is 15 bit for the first level table and 10 bit for the second level table.
For the first level table `K` has a fixed size of 6 bit. The chosen size results in mapping 64 byte of contiguous memory to the same key. That solves the issue with aligned instructions. In addition, when the lookup in the first hash table indicates that no breakpoint is set, this is also true for consecutive addresses in the same 64 byte range and hence the lookup can be skipped for these subsequent addresses. Another benefit is, that the amount of different memory locations accessed when executing a range of contiguous machine instructions is reduced by up to the factor 64 (in case of 1 byte instructions).
We chose 11 as size of `K` for the second level table. This allows to calculate the key for the second table by shifting the key of the first one by 5 bit to the right (i.e., dividing by 32).
During our benchmarks the chosen method for deriving the keys did not cause collisions and the lookups were fast enough, hence we did not evaluate further sizes or other methods for key derivation.

Disadvantage

The disadvantage of our approach is the need to pre-compute the complex hash maps and their dependency on the current context. Our approach is in fact a trade off between the time for the breakpoint check and the time to generate these structures. As the context changes very frequently in a computer system, we take actions to amortize the pre-computation time over a longer time period. The implementation details of this approach are covered in the next section.

5.6.2 Pre-Computed Introspection Module Instance Sets

The context of the guest changes very frequently, every task switch for example yields a new context. However, the contexts typically reoccur. When the OS switches to another task, it will very likely switch back to the previous task after a while.
We take advantage of this by dividing the interface used by the OS I-module into two parts. The first part builds and modifies instance module sets. This must be used when a new context occurs or an old one needs modification. When an I-module instance is added to

a context, the structures for the Fast Breakpoint Search must be updated to include the breakpoints of the new I-module relocated by the offset specified in the instance. As the speedup structures are rather complex and huge, this is time consuming.
However, the second part of the interface only instructs the Full System Simulator Interface to switch to another introspection context whose speedup structures are already build. As such a switch only implies changing a pointer, it is very fast.

We moved the cost of generating speedup structures to the creation of I-module Instance Sets. The result in respect to a guest computer system: It is rather expensive to create a task which modifies the context, i.e., an I-module is loaded to analyze the program, but the cost of a task switch is low. We expect the latter to occur *much* more often.

5.7 Full System Simulator Interface Implementation: QEMU Specific

QEMU is a Dynamic Binary Translation based full system simulator which we discussed in detail in Section 2.1.3. We integrated the Full System Simulator Interface into QEMU's code while our primary objective was to keep the performance impact low.
To be able to load and unload I-modules interactively during runtime, we extended the control interface of QEMU called *monitor* to provide commands which instruct the Full System Simulator Interface to load or unload I-modules from the host hard disk.

The remaining simulator dependent parts of the interface are the access of guest state (§ 5.7.1) and the implementation of breakpoints (§ 5.7.2). We end this section with a discussion of a QEMU specific problem caused by the design of the Tiny Code Generator (TCG) (§ 5.7.3).

5.7.1 Guest State Access

The guest state which the Full System Simulator Interface must provide consists of the value of the registers and the RAM described using virtual addresses. We handle these two locations differently and discuss them in separate sections.
A common attribute of the accessed data is its dependence on the current guest CPU state (§ 4.3.1). For example a virtual address may refer to different physical memory locations depending on the page table currently used. Therefore all interface functions accessing the guest state require the current CPU context. This can be provided by the breakpoint handlers as the guest CPU state is part of the information which is passed to a handler in case of a breakpoint.

Register Access

The register content can be found in the `CPUArchState` structure which represents a CPU instance in QEMU. As we access this content directly in the individual I-module without calling any interface functions, this part of the interface is not implemented for

QEMU. As the IP is updated lazily by QEMU (§ 2.1.3), care must be taken to ensure that the value seen by the breakpoint handlers is up-to-date. So we force an update of the IP prior to calling the breakpoint handlers.

Memory Access

The QEMU dependent access to the guest RAM consists of only three different functions:

1. `vp_copyGuestMemory(...)`
2. `vp_getGuestMemoryPtr(...)`
3. `vp_copyToGuestMemory(...)`

The first two functions implement read access to the guest, they differ in the fact, that the "copy" function guarantees to return a private copy of the guest's memory content. The other function may return a private copy or a pointer to memory only valid during the breakpoint handling. The information on the semantic of the returned pointer and whether the memory access succeeded is passed by flags.
Differentiating between these two request types generally avoids creating unnecessary copies, if only a pointer to memory is required. However, it is often not possible to avoid copying memory, if the requested virtual memory crosses page boundaries. The single virtual pages of the guest are represented by arbitrary locations in the guest's "physical" RAM. Hence pages located contiguously in the guest's virtual address space are likely to be scattered in the host's virtual memory.
Currently, our implementation for QEMU always copies the memory because of some issues possible when using direct pointers to the guest memory in QEMU's implementation. The third function implements write access to the guest. Writing to a guest is not a primary aim of our approach, but we intended to demonstrate that it is feasible.
The implementation of these three functions is simple as QEMU provides the `address_space_read/write(...)` functions which allow access to the physical representation of guest memory. As the access through the interface is done by using the virtual addresses, we must translate these addresses to physical ones. QEMU contains a function called `cpu_get_phys_page_debug(...)` which provides this feature.

5.7.2 Tiny Code Generator Based Fast Breakpoints

A simple way to integrate our breakpoint check into QEMU would be to insert a helper function before each translated instruction. This function would check for a breakpoint hit and call the respective breakpoint handler. The disadvantage of this approach is that we would check the same addresses over and over again if we execute a TB several times. These unnecessary checks can be avoided by moving the breakpoint check into the translation process. We only insert a call to a helper function if the respective IP is currently a breakpoint location. As the translation result is memorized, the performance loss due to the check is reduced further.
When using this approach, some new issues arise caused by the caching and the construction of the TCG cache itself. We discuss them in the following two sections.

Interrupting the Execution of Translation Blocks

The breakpoint handlers are called while the guest CPU is executing a TB. This means that the decision to change the current introspection context, and with it the breakpoint locations, is made while executing a TB containing breakpoints.
As mentioned earlier (§ 2.1.3) QEMU may rebuild TBs in several situations, e.g., to calculate the guest IP at a certain location. This fails if the new translation of a block results in different host code. If we change the context instantly, this might happen.
We solved this issue by two actions:

1. We postpone the introspection context switch until the execution of the guest instruction, at which the breakpoint is set, is finished.

2. We end every TB, if we insert an instruction prepended with a breakpoint handler.

As a result we are able to guarantee that the simulator is not inside the execution of a TB once the breakpointed instruction has been executed and we are hence no longer prone to issues caused by changing the introspection context within a TB.
Postponing the context switch after the execution of the instruction does not harm as the switch is still executed before the next check for a possible breakpoint is necessary.

Introspection Context Switch

When the OS I-module instructs the interface to switch to a different introspection context we also have to cope with cached TBs. They were translated based on the previous context and may hence:

1. Contain breakpoints which are no longer valid.

2. Miss breakpoints which should be added to the TB.

An easy solution would be to flush the entire TB cache on every introspection context switch. However this implies a huge performance impact as it would decrease the time a TB can remain inside the cache heavily.

To avoid such a flush, we introduce a version counter increased with every introspection context switch. Whenever a TB is build, the current value of this version counter is memorized within the block. We also record whether we inserted a breakpoint somewhere within the TB.
On an introspection context switch the version counter is incremented and the chaining of the current TB is destroyed. This forces the program flow within the simulator to call a function searching for the next TB to execute.
We modified the function responsible for locating cached TBs. Before passing on a located TB, its version counter value is checked against the current value. On a mismatch the following actions are taken:

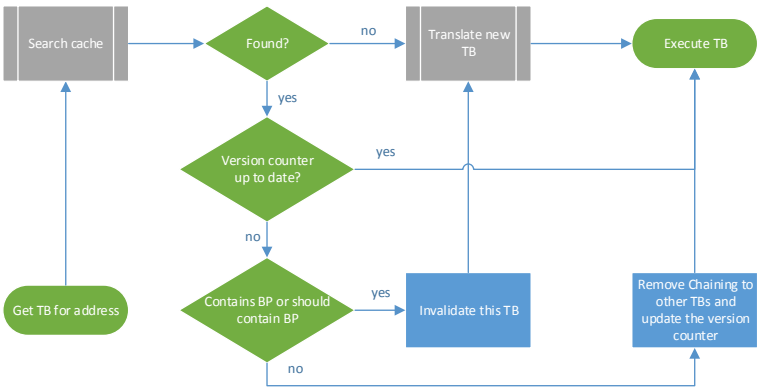

Figure 5.6: Modified program flow when searching for a TB in QEMU. This process is only executed when the current TB is not directly chained to another TB.

1. If the TB contains a breakpoint or the address range covered by the TB should contain a breakpoint according to the current introspection context: The TB is invalidated and hence it is newly translated by QEMU.

   We delete all TBs which contain breakpoints as it is difficult and time consuming to decide whether the breakpoint is still valid for the current context.

2. Otherwise: The version counter value of the TB is updated, but the chaining of this TB is invalidated to allow checking of the TBs possibly jumped to by this block.

The new process of searching for a TB is depicted in Figure 5.6. By using this approach we must only invalidate a rather small amount of translation blocks and the cost during normal runtime is only a compare operation of two integers when selecting a TB from cache. The destruction of the TB chaining is not permanent, but it is reestablished once a TB with a version counter already updated is executed again.

Multi-Core Support

QEMU implements multi-core support for guests by simulating the guest CPUs consecutively. As all CPUs share the same TCG cache, we might leak breakpoints set for one CPU into the execution of another CPU, which is likely to represent a different introspection context at that moment.
We currently solve this problem by incrementing the version counter on every switch between the simulated guest CPUs. Another approach would be to provide separate TCG caches for each CPU, but this would require substantial changes to QEMU's design.

5.7.3 QEMU Caused Restarted Instructions

TCG requires instructions causing device I/O to be located at the end of a TB for some configurations. However, it can happen that QEMU notices such an instruction only just

as it executes it. When the instruction is not at the end of the TB, QEMU builds a new block and restarts the execution there.
This would cause a breakpoint handler located in front of the interrupted instruction to be executed twice. We already discussed this problem and how we solved it (§ 4.2.3), but in this case the re-execution is not part of the normal execution flow within the guest ISA. It is only done because of the implementation of the TCG and therefore it should not be passed to the user.
As QEMU always restarts the instruction, these re-executions can be hidden completely from the I-modules by skipping them.

5.8 Conclusion

The implementation of the event-based introspection focuses on supporting a system using DWARF such as Linux, running as a QEMU guest. The implementation is divided into platform or format dependent code and universally usable parts to facilitate adapting to other environments. The Introspection Module Build System is solely implemented using macros and the GCC compiler. Breakpoint handlers and guest variable access functions are hidden by macros which expand to functions generated based on the RPI file. This file is created by a modular RPE implementation supporting plug-ins for different debugging symbol formats. Currently only a plug-in for DWARF exists.
The Full System Simulator Interface implementation is build around a multi level breakpoint check structure which aims to reject most IP values in linear time independent of the number of active breakpoints. As these structures require time consuming pre-calculations the introspection context is represented by pre-calculated I-module sets which avoid rebuilding the structures on every introspection context switch. QEMU allows further reduction of the overhead of breakpoint checks by integrating them into the DBT process. The conditional integration of helper functions into the TBs required modifications to QEMU's process of TB caching and the introduction of a version counter for the TBs.

Chapter 6
Evaluation

In the previous chapter, we described the implementation of an event-based introspection approach. This chapter focuses on evaluating that implementation and on discussing limitations of the approach used. At the beginning we present some created Introspection Modules (I-modules) and benchmarks of their execution in Section 6.1. The limitations caused by issues related to debugging symbols are discussed in Section 6.2.
Section 6.3 covers issues with unmapped pages when using virtual memory and solutions to cope with them. Considerations on how to handle the boot process within the introspection are examined in Section 6.4. The chapter ends with an analysis of the applicability of our approach for closed-source OSes in Section 6.5.

6.1 Implemented Introspection Modules

We used our introspection tool to build I-modules for several scenarios of event-based introspection for a Linux guest. We implemented a basic OS I-module (§ 6.1.1) and an I-module for a simple self-written Kernel Module (K-module) (§ 6.1.2). In order to demonstrate the ability to introspect user programs, the implementation of an I-module for a user program is discussed (§ 6.1.3).

6.1.1 Introspection Module for the Linux Operating System

The OS I-module for the Linux kernel must provide the introspection context. We implemented only a minimal set of breakpoints required for this functionality. The chosen breakpoint locations are:

Start of Function `wake_up_new_task(...)` This function is called when a new task is created. We extract the Process Identifier (PID) and the name recorded in Linux's `task_struct`. However, this name is not the name of the process which might be loaded shortly afterwards, in case the `execve` system call is used to replace the binary image of the process. Instead it is still the name of the program

which executed the `fork`-call, as the breakpointed function implements a part of the `fork`-procedure.

If we are interested in the process name of the task which might be loaded after the `fork`, we need to add a breakpoint into a function implementing the `execve` system call. However, we did not require this information in our tests.

Start of Function `exit_notify(...)` This function is called when a task is deleted; again we extract the PID and the name.

Start of Function `__switch_to(...)` This function is called on every task switch. We only extract the PID of the task we switch to.

These three breakpoints are sufficient to track which userland program is currently running and they are equal to the locations we chose in the performance analysis of the debugger based introspection approach (§ 3.3). The following sections discuss a performance measurement and cover the validation of the introspection results.

Performance Comparison

In the analysis chapter we argued that the performance is an important attribute for an introspection approach. In the following we discuss the slowdown caused by our introspection approach and compare it to the implementation using a debugger (§ 3.3). In order to be able to compare the results, we use the hardware and software known from the debugger based benchmark; see Table 6.1 for details. If not stated otherwise, all measurements were executed five times and the average values are given. The results of our approach are named with "vpanopticon" in the figures.

Property	Value
Host OS	Ubuntu, Kernel 3.2.0-52-generic
Host Hardware	Intel Q6600 @ 2.40 (64 bit), 4 cores GHz, 7 GiB RAM
Host GDB	GNU GDB (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Guest OS	Ubuntu, Kernel 3.5.0-23-generic
Guest Hardware	AMD64, 1 core, 1024 MiB RAM

Table 6.1: Details on the host system running QEMU and on the simulated guest system.

The workload for the measurements is a minimal Linux kernel build using one (j1) or two (j2) threads. QEMU was configured to use the `icount` extension in most test cases, however we also ran some tests with this extension disabled. Figure 6.1 depicts the different runtimes. The slowdown of our approach compared to the not introspected runs was always below 10 percent, details can be found in Table 6.2.
When the `icount` extension was disabled, the performance of the GDB based introspection further decreased. When about half of the output generated during a kernel build was printed by the guest, GDB consumed almost all free RAM of the host system and the execution was stalled. We assume this to be caused by a memory leak in GDB and hence

we could not execute this benchmark to the end. So the runtime we specify is the time between the start and last time we received a line from the guest concerning the build process.

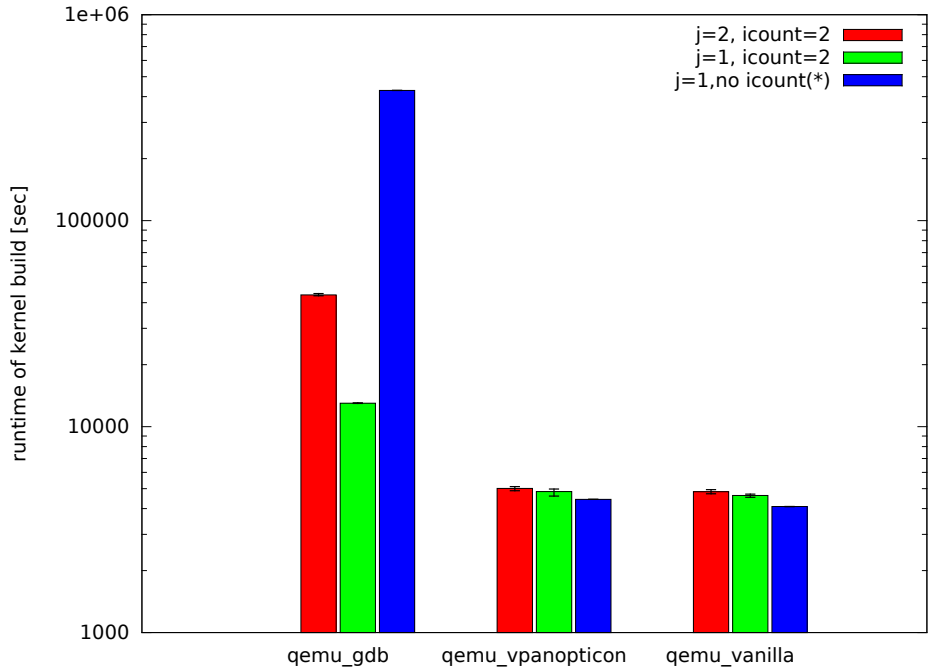

Figure 6.1: The bar chart depicts the wall-clock time necessary for a kernel build on the guest when using the named introspection approach. The error bars represent the minimum and maximum values. The runs with disabled `icount` were only executed once and we interrupted the execution of the GDB based introspection before the end.

The additional slowdown for the GDB based introspection can be explained by the already huge slowdown of this approach. GDB inserts a breakpoint which is called at every task switch and every call to this breakpoint takes about 40 milliseconds (§ 3.3). As the time of the host is passed directly to the guest when not using the `icount` extension, GDB already consumes a relevant part of the guest process's time slice. Hence more task switches occur and with that more events must be handled during the overall compilation.

As the execution of GDB's guest is neither deterministic nor completely isolated from the host system, even when using the `icount` extension, the number of events differed between the two introspection approaches. The number of events measured by the GDB based approach was increased by about factor 1.04 (j1) and 1.34 (j2) compared to our approach. To verify the difference in the measured numbers is not caused by a fault in our approach, we compared the results of our approach to the information put out by a modified kernel. The verification is discussed in the next section.

		Runtime (s)	Runtime Factor	#Events/s
QEMU,	**j1, no icount**[(i)]	**4092**	**1.00**	**—**
QEMU + GDB,	j1, no icount[(i)]	> 428943	> 104.82	—[(ii)]
QEMU + vpanopticon,	j1, no icount[(i)]	4433	$\simeq 1.08$	$\simeq 50.99$
QEMU,	**j1, icount=2**	$\simeq$ **4634.0**	**1.00**	**—**
QEMU + GDB,	j1, icount=2	$\simeq 12991.2$	$\simeq 2.80$	$\simeq 7.07$
QEMU + vpanopticon,	j1, icount=2	$\simeq 4838.4$	$\simeq 1.04$	$\simeq 18.32$
QEMU,	**j2, icount=2**	$\simeq$ **4828.8**	**1.00**	**—**
QEMU + GDB,	j2, icount=2	$\simeq 43584.2$	$\simeq 9.03$	$\simeq 8.79$
QEMU + vpanopticon,	j2, icount=2	$\simeq 5010.2$	$\simeq 1.04$	$\simeq 57.15$

(i) These benchmarks were ran only once because of the extreme runtime for the GDB based approach.
(ii) The number of events cannot be given reliably as we interrupted the benchmark.

Table 6.2: Runtime comparison for compiling the Linux kernel on a guest system using different introspection approaches. Each test was run 5 times and the mean values are given. The time is measured in host time, i.e., wall-clock time.

Validation

The processes necessary for the introspection such as parsing debugging symbols and modifying the full system simulator are complex and therefore prone to bugs. In order to minimize the chance to miss any bugs, we additionally built a Linux kernel containing `kprintf(...)` instructions at the locations we use for our introspection. The same information accessed by the OS I-module was also printed and the output was forwarded to a log file on the host system.

The modifications to the source code of the kernel caused the compiler to optimize-out the pointer to the `task_struct`, which is needed at the breakpoint in `exit_notify(...)` and which was accessible in the unmodified kernel previously. This is a general issue with debugging symbols and will be discussed later within this chapter. Instead of relocating the breakpoint, we solved the issue in this case by using a compiler pragma[1] to disable optimization for this function.

The recorded logs for a boot process matched the output of our I-module exactly. Though this validation cannot be done for the unmodified kernel we used in the previous measurement, this demonstrates our approach to work in general. To verify the introspection results of the unmodified kernel as well, we tested the recorded events for the kernel manually by starting programs and ensuring that the intended events were actually issued.

[1]The function is wrapped by the pragmas `#pragma GCC push_options`, `#pragma GCC optimize ("O0")` and `#pragma GCC pop_options`.

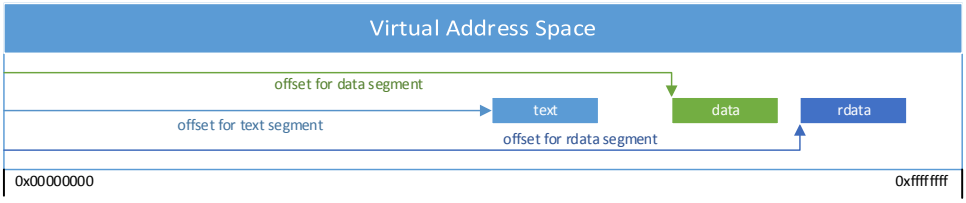

Figure 6.2: When loading a Linux K-module into the memory, its logical segments can be moved relatively to each other. Hence a single offset is insufficient to describe the calculation of the actual addresses in the memory image.

6.1.2 Introspection Module for a Linux Kernel Module

The Linux kernel supports loading of K-modules during runtime. To demonstrate the ability to introspect code provided by such K-modules, we wrote a simple K-module which outputs some global and local variables to the system log periodically. The corresponding I-module adds a breakpoint at the function responsible for printing the values within the K-module and it extracts the values from the guest.

The OS I-module had to be extended with a breakpoint at the start of the `add_notes_attrs(...)` function. It is called inside the guest OS whenever a K-module is loaded. It was necessary to modify our early implementation slightly as Linux K-modules are special compared to the binary file we expected:

1. Their code segment is split into several code segments to support loading of only the necessary code parts and to allow unloading of code parts no longer needed [40]. This allows reusing the memory of the code segment only required for the initialization, after a K-module was loaded. When unloading of K-modules is disabled, the segment containing the code for the unloading can be omitted during the load process.

2. The logical segments of the K-modules are moved *independently* from each other during the load process, as done while linking an executable file. Therefore a single offset is not sufficient to describe the modifications to the addresses recorded in the debugging symbols. Figure 6.2 depicts an example.

As some code segments are not loaded all the time, the first issue must be addressed by providing different I-modules for each code segment. By using different I-modules, an introspection context, where some segments are loaded and some are unloaded, can be represented. In this case, a single offset is still sufficient to describe the relocation of the breakpoints within a single I-module, as there is only a single code segment per I-module.

The second issue required small changes to the Full System Simulator Interface and the way addresses are encoded for such I-modules. As discussed in the implementation chapter (§ 5.2) the RPE implementation allows to refer to addresses by IDs representing an address by a logical section and the offset within it.

When the RPE generates Location Description Instructions for Linux K-modules, instructions accessing guest addresses do not longer add an offset, provided by the I-module instance, to a hard-coded address. Instead the instructions contain only a hard-coded index for a table containing the addresses. This table is created for each I-module instance. It is generated based on a source table contained in the I-module when the Full System Interface creates an instance. This source table describes every address, required in any Location Description Instruction, by the section name, in which it is contained, and by the offset within the section. To translate this information to absolute addresses, it is necessary to know the starting address of each segment used such as `data` or `rdata`.

When loading I-modules using such an advanced address relocation, the OS module must provide that information. The breakpoint we chose in the Linux kernel allows to access this information via accessing the `sect_attrs` structure attached to module information created for the loaded K-module.

The DWARF format does not offer information on the logical segment an address is contained in. Therefore the DWARF plug-in for RPE must extract this information by matching the origin of the address value in the debugging symbols with the relocation information recorded in the ELF file format.

Using the described modifications we were able to introspect the K-module and extract the variable values. In this case, validating the correctness of our introspection results was easy as they were by design printed to the guest's system log as well.

6.1.3 Introspection Module for a Simple Userland Program

Support of introspection of userland programs is an interesting feature of an introspection approach. We decided on introspecting a simple self-written program which reads user input and outputs the read text to the terminal.

To allow the OS I-module to determine when our program is started, a further breakpoint could be added at the function implementing the `execve` system call. However, for our testing purpose it was sufficient to use a *marker program* to fork and start our program. As we already catch the fork and the name of the forking program, we can tell implicitly which program will be started and which PID it will have. A simple Linux program usable as a marker is `sudo`.

This is only acceptable in this context, as in general a different program could be started with `sudo`. Additionally, the introspection context does not correctly represent the state in the guest machine for the short time the I-module is attached to the forked `sudo` process and the marker program has not yet replaced itself with our self-written program. However, it was sufficient for our tests. Additionally, using a marker allows running several instances of a program, some of them marked to be introspected, some of them not.

The remaining requirement to create an instance of the I-module for our program is the offset for the program's addresses. In our case this offset is 0 as simple Linux

programs directly contain the addresses they are loaded at and hence no modification of the addresses is necessary.

As the OS I-module is informed of every task switch and we provided a solution to determine the PID of the process executing the program to introspect, it can switch the introspection context to a task specific context, while our program is executed. When the guest OS schedules a task which is not a target for introspection, the default introspection context is used. This default context contains only the kernel dependent I-modules such as the OS I-module and other I-modules representing K-modules.

Using this combination of the modified OS I-module and the I-module for the process, it is possible to intercept the text inserted by a user and to even modify the text. As the modification of guest state was not the aim of this work, we did not provide a general solution. It is currently only possible to modify variables located in RAM.

6.2 Debugging Symbol Related Limitations

The previous benchmarks demonstrated that our approach can be used to introspect guest systems and their applications while providing a high performance. However, our approach uses debugging symbols to bridge the semantic gap and hence all limitations related to debugging symbols in general are also limitations to our approach. Especially when executables are build with a high optimization level, the created debugging symbols are not always sufficient to access every variable at every location (§ 6.2.1). Automatically translating a logical breakpoint location in the source code to an IP value can be difficult or even impossible depending on the provided debugging symbols (§ 6.2.2).
In theory the debugging symbols already discussed (§ 2.3) can contain arbitrary complex Access Paths which describe locations of variables, hence the translation to C code can become difficult (§ 6.2.3). In the C/C++ programming language macros create further issues as they are expanded before the compilation process and hence are not directly available to the compiler during generation of debugging symbols (§ 6.2.4).

6.2.1 Optimized-Out Variables

Some variables, present within the source code of a function, are not necessarily accessible at all machine code locations representing the function. When using optimized code, the variables' values might not always exist in memory or registers during the execution of the machine code representing the function.
An example for variables which are candidates for this issue are function parameters only used in the upper part of a function but whose value should be accessed by an introspection tool at the end of the function. Following the last access to a variable's value, according to the source code, the compiler can reuse the location, e.g., a register, to memorize other values. As the variable's value is overwritten, it cannot be accessed at a subsequent position.

Nevertheless, it is possible to generate debugging symbols allowing to calculate the values of optimized-out variables originating from other variables which are still present, e.g., by generating instructions on how to calculate the value of a function parameter from the stack frame of the current function's caller [28].

Referring to an analysis of a `cc1plus` binary built using GCC, Jelínek [28] claims that only 31% of the variables and parameters were accessible in all code bytes generated of functions where they should be accessible semantically. To cope with optimized-out values in introspection, it may help to set several breakpoints inside a function to access variables at positions where they are still accessible and to memorize their value temporarily until the breakpoint originally intended is hit.

Besides waiting for further improved debugging symbols generated by compilers, modifying the binary is an alternative. This contradicts the initial aim of not influencing the guest at all, however, it might still be an acceptable solution for different scenarios as the proposed modifications still allow running the guest system with or without introspection or on real hardware. The compilation process can be influenced in several ways to produce a binary which is unlikely to be prone to optimized-out variables on certain locations. The following list contains some possible modifications:

1. The optimization can be disabled globally. While this allows to easily add breakpoints at all locations, it also influences the performance and behavior of the executable heavily.
2. To affect only small parts of an executable, the optimization can be disabled only for a relevant function by using compiler specific pragmas.
3. In order to further reduce the amount of code built without optimization, only a call to an unoptimized function can be inserted at the intended breakpoint locations. The introspection breakpoint could be set at the inserted function and the function parameters could be accessed.

As none of these modifications uses hypercalls relying on invalid instructions [25], the resulting binary is still a valid executable file runnable on non-simulated hardware.

6.2.2 Locating Breakpoint Addresses

Debugging symbols provide a mapping between source code locations and machine code. This mapping is necessary to determine breakpoint addresses for logical breakpoint locations in the source code. While it is easy to use this mapping for unoptimized code, optimization causes several issues. The basic problem is that the optimized machine instructions do not necessarily represent the instruction flow exactly as defined in the source code.
Reordered instructions raise the question, whether a breakpoint location can be found where all variables of the running program have the value, they semantically should have, according to the source code. For example "`a = 2; b = 3`" can in general also

be represented as "`b = 3; a = 2;`" in the generated machine instructions without affecting the correctness of the code. However, inserting a breakpoint after the assignment of `a`, but before the assignment of `b` is impossible in this case.
We currently bypass this problem by using function starts as breakpoints for optimized code in the implementation of our modules. However, setting breakpoints only at function starts is not sufficient in general, hence we allow semi-automatic definitions of breakpoint locations, too.
The user can request listings of the mapping of source code lines to addresses within the instances of a function. By using other sources of information such as knowledge on locks, memory barriers or the exact binary instructions of the function, the user can decide on breakpoint addresses for a single logical breakpoint. Because of the huge difficulty to select breakpoint addresses by hand this is an important subject for further improvements. For partially inlined functions [33] the search for breakpoint locations is even more difficult as the instances only represent parts of the function. Hence not every instance necessarily contains a breakpoint representing a certain logical breakpoint in the function.

As for optimized-out variables, it is also possible to cope with this issue by modifying the locations of interest to provide code and debugging symbols simple enough for us to calculate breakpoint addresses automatically.

6.2.3 Translation of Location Descriptions

In order to support debugging of optimized executable files, modern debugging symbol formates provide a very flexible description for the locations of variables. The DWARF format for example uses byte code evaluated on a stack machine. While the code was rather simple for all cases which occurred during our tests, it can be very complex in general.
It is an open question whether translating every location description for every debugging file format into hard-coded Location Description Instructions is possible in general.

6.2.4 Macros

The C/C++ programming language uses macros which are expanded to pure code before the actual compilation. Macros can be used to mask variables or to insert code at different locations. As the macros are expanded by the pre-processor before the compilation, the debugging symbols cannot represent them inherently. However, DWARF for example provides explicit support to represent these macros in the debugging symbols [18].
Nevertheless, we currently do not support macros at all, as the available debugging symbols generated by GCC for the Ubuntu kernel did not include this information[2]. The lack of support for macros does not cause general restrictions, but instead it "only" results in a bit more work writing a RPC file. If for example a macro is used to hide a variable name in a function, the macro must be resolved by hand and the resulting variable name must be specified in the RPC file.

[2] It is however possible to instruct GCC to add this information using the `-g3` option [53].

6.3 Unmapped Memory

Virtual memory can be used to overcommit RAM. Not all pages have to be located in the physical RAM concurrently, as the OS loads pages not present in case they are accessed.

When accessing a guest variable using an introspection approach, the variable, or parts of the information required to find the location of the variable, might be located on such a page not currently mapped. Accessing the variable is not possible without issuing a page fault to force the guest OS to map the relevant pages. However, by doing so the guest's state would be altered and this is undesirable in general.

In the following we discuss this issue for an OS with swapping disabled. Afterwards we cover the consequences when it is enabled.

6.3.1 Swapping Disabled

When swapping is disabled, the only cause for a not mapped page is that it was never read or written. This implies that the contained variable currently has a value which can be determined by knowledge of the executable format and the source code. Therefore the inability to access the variable content does not imply a miss of information.

To cope with this issue in an I-module, the flags returned when accessing a guest variable must be checked to ensure the access was successful. If it failed, the User Defined Code must contain instructions to use a default value provided by the user.

The situation is slightly more difficult for Access Paths which involve resolving of pointers. For example a pointer (`p`) might be modified to point to a new target and hence the pointer would be accessible. However, the target of the pointer might be a global variable (`a`) which has not yet been read or written and hence its page might be unmapped.

Therefore using the Access Path "`p=>*`" to access the value of the pointer's target can fail. To determine the correct value in this case, all possible global targets for the pointer must be determined using the source code of the guest program. The addresses of the possible targets can be accessed using additional Access Paths (e.g., "`a`"), afterwards they can be compared to the current target of the pointer ("`p=>`"). The correct value for the path would hence be the default value of the matching global variable.

If none of the addresses matches, the target resides in dynamically allocated memory such as the stack or the heap. Hence it must have been initialized during runtime (and thus would be accessible) or it contains the default value provided for newly mapped pages (e.g., filled with zeros). Therefore it is also possible to determine its value.

While this seems to be complex in general, it is often unnecessary to implement the methods described above, as a user is likely to be interested in variables which have been read or written by the guest shortly before. If this is true for all variables used in a path, they are guaranteed to be located on pages already mapped and hence the access using the Access Path cannot fail (assuming all contained pointers refer to valid memory locations).

6.3.2 Swapping Enabled

When swapping is enabled, a page already modified can be removed from RAM and stored on the hard drive. As this breaks the assumption that non-available pages only contain default values, the previous solution cannot be used. However, as a user is likely to be interested in variables which were modified shortly before, it is likely that the pages containing them are currently located within the RAM. In addition, important parts of OSes' code and data are not swapped out typically.
In general it is not possible to rely on any of the assumptions given above and hence the handling of an occurrence of a non-mapped page must be supported. There are three possibilities to cope with such a situation:

1. For some measurements it might be tolerable to not be able to provide the value of a variable, if at least no wrong values are read. In this case no additional actions must be taken, failure of accessing a guest variable is already signaled to the User Defined Code.

2. As inaccessible variables are sometimes problematic, a general solution is necessary. As the page's content is located somewhere on the hard drive, it is possible to build code which is able to parse the information contained within the guest OS and to read the page content from the guest's virtual hard drive. However, this is a very complex task as it is OS dependent and it further involves the access to the hard drive and requires to understand the file system structures involved.

3. If altering the guest is tolerated, this problem can be solved more easily by triggering page faults inside the guest and letting the guest OS provide the page's content. While this is much easier to implement, it might lead to unexpected guest behavior because of page faults which would not occur normally.

For all our tests, we used a Linux system without any swap space and hence swapping was disabled. Implementing and testing of the possible solutions given above was out of scope of this work.

6.4 Boot-Up Related Considerations

Until now we argued on the assumption that the OS is already running inside the guest. In this situation it is correct to consider the OS I-module to be part of every context for ISAs such as AMD64. As an OS is typically loaded and started by a bootloader, it is in theory not correct to load the OS I-module at boot time. At least for the AMD64 ISA, bootloaders are located at low addresses whereas an OS typically resides at high addresses, hence their address ranges did not overlap and this did not cause an issue in our case.
In general, a bootloader might use code ranges to be overwritten later with OS code. To avoid issuing of wrong breakpoints, the OS I-module must be deactivated temporarily during the runtime of the bootloader. This can be implemented by initially loading an I-module for the bootloader which replaces itself with an instance of the OS I-module as soon as the loading of the OS is finished.

A further solution is to modify the Full System Simulator Interface, to only issue breakpoints in the CPU mode which the OS uses. However, the bootloader might also run partly in this mode and hence the solution could fail in this case.

6.5 Support for Closed-Source Operating Systems

The implementation of our approach was targeted at Linux, however, introspection is not only a topic for open source OSes and programs. This section discusses the differences for closed-source software.

An advantage of closed-source OSes is that they typically provide a callback interface for a lot of important kernel events such as process creation to allow implementation of foreign K-modules. These callback interfaces are likely to be good and stable locations for breakpoints representing important changes within the guest.

However, even if the introspection only aims at introspecting guest programs for which the source and the debugging symbols are available, it is still necessary to retrieve the guest events, related to tasks such as the change between different user processes, from the OS.

In the following we consider two different scenarios (i) the worst case when no debugging symbols are available and (ii) the case when only the source-code is unavailable.

6.5.1 Assumption: No Debugging Symbols, No Source Code

When no or insufficient debugging information is available, the automatic creation of RPI files is impossible for our approach, but it would be possible to handcraft a RPI file and use it to build an OS I-module. However, writing such a RPI is very difficult without knowledge of the binary as it is provided by debugging symbols and a handcrafted RPI file would only match a special version of the OS kernel.

For OSes such as Microsoft Windows which allow writing of K-modules, a solution might be to create a K-module which uses the provided callback interface to receive the desired kernel events. As the source and the debugging symbols are available for this K-module an I-module can be generated automatically. The drawback of this approach is that the guest system is modified by inserting the K-module.

In addition, it is still necessary to obtain the memory location of the K-module as this is required by the Full System Simulator Interface to determine the actual locations of the breakpoints. When using an I-module for a kernel this is normally not necessary, as the start address of the kernel is often constant and contained in the kernel binary, but K-modules are normally not guaranteed to be placed at a certain address.

Solutions to propagate the start address depend on the individual OS and situation, but a simple solution would be to include code into the K-module to print the start address within the guest.

6.5.2 Assumption: Complete Debugging Symbols, No (complete) Source Code

As this situation provides all information available in the previous case, all methods described above can be used. Because debugging symbols are available, the transformation from RPC to RPI files is possible.
Nevertheless, as the source code is not available, it is difficult to know where to put breakpoints representing the kernel events. Functions containing code where breakpoints should be located might be determined by their (hopefully meaningful) names or by additional documentation. However, the breakpoints can only be set at the start of the function. To select a precise position within the code, the source code line must be specified and this is not possible without knowing the source code.
This limitation can be circumvented by selecting the IP values by looking at the binary file, but this is again difficult and must be done for every version of the kernel.

6.6 Conclusion

The introspection approach presented proved to provide a good performance compared to a debugger based approach and even introspection of guest programs is possible. In all benchmarks the slowdown caused by our implementation was below 10%.
However, in case of executables built using optimization, debugging symbols do not always provide sufficient information or it is difficult to use them. Therefore the process of building I-modules sometimes requires to manually influence the addresses where breakpoints are set and to search for locations where a necessary variable is not optimized-out.
Accessing virtual memory can result in difficult situations, in particular when using a guest OS with swapping enabled. Though the memory of pages, accessed shortly before by the guest, is unlikely to be swapped out, it is impossible to guarantee this in case of swapping enabled. Thus it is necessary to provide a way to access swapped out memory locations.

Most issues, preventing the building of I-modules fully automatically or requiring more complex User Defined Code, can be more easily circumvented by permitting modification of the guest or the programs run within it.
If small modifications are acceptable, this can offer a rather simple solution for building I-modules automatically instead of intervening manually in the build process — even if the executables are compiled with optimization enabled globally.

Chapter 7
Conclusion

Full system simulators are useful for analyzing processes and events within a computer system. As the complete guest system is simulated in software, the state of the guest can be accessed at an arbitrary temporal resolution. However, most information in a computer system is not held in locations which can be known by referring to the ISA. To acquire large parts of the high-level information such as the content of variables used in guest programs, it is necessary to bridge the semantic gap between the low-level information, a full system simulator can access, and the high-level meaning of this information within the executed program.
Existing introspection approaches focused on the analysis of (RAM) snapshots of computer systems. While this can yield a lot of useful information for a single point in time, it is insufficient for continuously introspecting a running system. In respect to performance, it is impossible to use these approaches as they do not indicate when the guest state must be analyzed and hence they need to be executed after each change of the guest state — that is after each write to the memory of the guest, as most of these approaches employ RAM images. In addition, there are problems with partially written structures.

We proposed an introspection framework capable of introspecting a guest system continuously. The concept of an introspection context allows to introspect guest applications and other code not mapped into the address space constantly. Depending on which code parts and programs are represented in the currently active virtual address space, a set of Introspection Modules (I-modules) is used. Each I-module represents an executable file such as the kernel or a program. They contain breakpoint information and code to extract guest variables and structures.
The framework provided to build such modules, allows to hide most difficult aspects of event-based introspection from the user, such as the possible existence of multiple breakpoint addresses for a single logical breakpoint. The task of writing an I-module requires only to write C code and to provide a configuration file defining the guest structures used in the code. The author of an I-module does not need to attend to the actions required to retrieve guest variables from the simulated system, as guest variables are accessed by symbolic names.

The code necessary to perform the accesses to the guest is generated automatically based on debugging symbols. However, for highly optimized executable files the current implementation occasionally requires additional help from the user to decide on breakpoint locations. In addition, accessing guest variables can be difficult as they are not necessarily accessible at all locations to be expected according to the source code due to optimization by the compiler.

Using the provided framework, we were able to build several I-modules which demonstrated the ability to inspect the guest OS and its programs. The benchmarks of our implementation in combination with the full system simulator QEMU indicated only a minor runtime increase below factor 1.1 for observing task related changes such as task switches of a Linux kernel. This offers a huge performance advantage over an alternative approach we implemented merely using a debugger which increased the runtime by factor 2.8 to 104.8.

7.1 Future Work

The current implementation of our approach contains some limitations not resulting from the basic design, but owing to the complexity of analyzing debugging symbols. In particular, the complete implementation of plug-ins for the Runtime Position Extractor (RPE) component for multiple debugging symbol formats would require a considerable effort.
Parsing of debugging symbols has already been implemented in debuggers and especially GDB contains the concept of agent expressions which also requires to pre-calculate the actions necessary for accessing guest variables. Therefore an important part of further development is to verify if and how it is feasible to base the RPE plug-ins on the code of debuggers.
As the complete process of parsing and translating debugging symbols is complex and the debugging symbols themselves could be wrong, an additional option to validate the introspection results would be beneficial. Although it is possible to validate the results manually, some sort of automatic checks would be much easier to use.

While the performance of the implementation is quite acceptable, it could be improved further by approaches such as caching the memory accesses to the memory of the guest and the required address translations. However, as long as the number of introspection events per seconds is rather small as it was in our tests, such optimizations are not likely to improve the performance noticeably.

To widen the scope of applications of our introspection approach, some extensions could be implemented. *Memory breakpoints* are a functionality already known from debuggers and in the context of introspection they can be useful to capture modifications of variables, modified from a multitude of locations and hence a lot of code breakpoints would be necessary.

In particular, when observing the behavior of single guest applications, providing a *call-stack* and the logging of every executed instruction (and the IP) of this program could give assistance for reverse engineering of applications protecting themselves against being debugged.
Using a full system simulator which isolates the execution of the guest completely from the host, it should be impossible for the guest to realize it is currently under observation by our framework. Hence the program to be reverse engineered will not change its behavior as it would possibly do on detecting a debugger.

Appendix

Example: Runtime Position Configuration and Information Files

```
bins = ("[...]/vmlinux");

locations =
(
  {
    name  = "task_create";
    pos =
    {
      type           = "FuncLine";
      file           = "";
      function_name = "
          wake_up_new_task";
      function_line = 0;
    };
    vars =
    (
      {
        name = "pid";
        expr = "p=>pid*";
      },
      {
        name = "name";
        expr = "p=>comm*";
      }
    );
  }
);

structures =
(
  {
    name      = "task_struct";
    code_name = "task_struct";
    entries =
    (
      {
        name = "pid";
        code_name = "pid";
      },
      {
        name = "tasks";
        code_name = "tasks";
      }

    );
  },
  {
    name      = "list_head";
    code_name = "list_head";
    entries =
    (
      {
        name = "next";
        code_name = "next";
      },
      {
        name = "prev";
        code_name = "prev";
      }
    );
  }
);
```

Figure A.1: Extract of a RPC file which we used for the Linux kernel. A breakpoint is defined at the start of function `wake_up_new_task` and two variables of the created `task_struct` are marked to be used later. In addition to that we marked two structures and some of their members to be relevant.

```
1  // ### global structure support: ###
2  CREATE_GLOBAL_STRUCTURE_START(task_struct)
3      CREATE_GLOBAL_STRUCTURE_ENTRY(task_struct,pid,692,TYPE_INT,4,1,0,0,
           LITTLE)
4      CREATE_GLOBAL_STRUCTURE_ENTRY(task_struct,tasks,576,TYPE_UINT
           ,1,16,0,0,LITTLE)
5  CREATE_GLOBAL_STRUCTURE_END(task_struct)
6
7  CREATE_GLOBAL_STRUCTURE_START(list_head)
8      CREATE_GLOBAL_STRUCTURE_ENTRY(list_head,next,0,TYPE_UINT,8,1,0,0,
           LITTLE)
9      CREATE_GLOBAL_STRUCTURE_ENTRY(list_head,prev,8,TYPE_UINT,8,1,0,0,
           LITTLE)
10 CREATE_GLOBAL_STRUCTURE_END(list_head)
11
12 // ### attribute access functions: ###
13 CREATE_ATTRIBUTE_ACCESS_FUNCTION_START(task_create,pid,TYPE_INT
       ,4,1,0,0,LITTLE)
14     CREATE_ATTRIBUTE_ACCESS_FUNCTION(task_create,0,pid,TYPE_INT
           ,4,1,0,0,LITTLE,
15         STACK_CREATE(stack, 5, TYPE_GUEST_MAX_ADDR) ENDL STACK_PUSH(
               stack, ADD(REGS(REG_7, 64), 8)) ENDL  STACK_PUSH(stack,
               REGS(REG_5, 64)) ENDL STACK_PUSH(stack, ADD(STACK_POP(stack
               ), 692)) ENDL  MARK_GMEM_AS_RESULT(STACK_POP(stack)) ENDL )
16 CREATE_ATTRIBUTE_ACCESS_FUNCTION_END(task_create,pid,TYPE_INT,4,1,0,0,
       LITTLE)
17
18 CREATE_ATTRIBUTE_ACCESS_FUNCTION_START(task_create,name,TYPE_INT
       ,1,16,0,0,LITTLE)
19     CREATE_ATTRIBUTE_ACCESS_FUNCTION(task_create,0,name,TYPE_INT
           ,1,16,0,0,LITTLE,
20         STACK_CREATE(stack, 5, TYPE_GUEST_MAX_ADDR) ENDL STACK_PUSH(
               stack, ADD(REGS(REG_7, 64), 8)) ENDL  STACK_PUSH(stack,
               REGS(REG_5, 64)) ENDL STACK_PUSH(stack, ADD(STACK_POP(stack
               ), 1120)) ENDL  MARK_GMEM_AS_RESULT(STACK_POP(stack)) ENDL
               )
21 CREATE_ATTRIBUTE_ACCESS_FUNCTION_END(task_create,name,TYPE_INT
       ,1,16,0,0,LITTLE)
22
23 // ### breakpoint sub-handlers:     ###
24 START_REGISTER_BP_HANDLER_ENTRY(task_create)
25     REGISTER_BP_HANDLER_ENTRY(0xffffffff8108a3c0, 0)
26 END_REGISTER_BP_HANDLER_ENTRY()
27
28 // ### breakpoint handlers:         ###
29 START_REGISTER_BP_HANDLERS()
30     REGISTER_BP_HANDLER(task_create)
31 END_REGISTER_BP_HANDLERS()
```

Figure A.2: Extract of a RPI file generated based on the RPC file from Figure A.1. The format is far from being easy to read by a human, hence we at least moved the Location Description Instruction into new lines. Both of these expressions start with `STACK_CREATE(...)` in this example. The advantage of this format is that it is rather easy to parse it using preprocessor macros.

Glossary

API Application Programming Interface. 21

BB Basic Block. 5, 6, 8, 10

DBT Dynamic Binary Translation. 4–6, 8–10, 62, 66

DIE Debugging Information Entry. 22–25, 53, 54

DWARF DWARF is a debugging file format commonly used in conjunction with ELF. 22–24, 26, 27, 49, 52–54, 66, 72, 75

ELF Executable and Linkable Format is a common file format for executables in Linux and other Unix-like OSes. 22, 72, 89

GCC GNU Compiler Collection. 54, 66, 74, 75

GDB GNU Debugger. 10, 26, 34–37, 68–70, 82

IP Instruction Pointer. 4–6, 8, 11–13, 23, 24, 35, 48, 53, 59, 60, 63, 64, 66, 73, 79, 83

ISA Instruction Set Architecture. 4–7, 9, 10, 14–17, 26, 32, 44, 49, 56, 61, 66, 77, 81

I-module Introspection Module. 27, 39–43, 46–48, 54, 55, 59, 61, 62, 64, 66, 67, 70–73, 76–79, 81, 82

K-module Kernel Module. 27, 51, 67, 71–73, 78

MMU Memory Management Unit. 7, 17

OS Operating System. 1, 2, 4, 10, 14–16, 18–21, 27, 29, 31–33, 35, 36, 38, 45, 47–49, 55, 61, 64, 67, 68, 70–73, 76–79, 82, 89

PDB Program Database is a proprietary debugging file format used in Microsoft Windows. 22, 25

PID Process Identifier. 19, 67, 68, 72, 73

QEMU Quick Emulator. vi, 2, 4, 8–14, 26, 27, 35, 36, 49, 58, 59, 62–66, 68, 70, 82

RAM Random Access Memory. 4, 7, 15, 16, 19, 21, 34, 36, 47, 62, 63, 68, 73, 76, 77, 81

RPC Runtime Position Configuration. 39–44, 46, 50–53, 75, 79, 86, 87

RPE Runtime Position Extractor. 41, 42, 49, 50, 54, 58, 59, 66, 71, 72, 82

RPI Runtime Position Information. 42–44, 46, 50, 52, 54, 57, 66, 78, 79, 87

SBT Static Binary Translation. 5

TB Translation Block. 10–14, 63–66

TCG Tiny Code Generator. 10, 11, 62, 63, 65, 66

Bibliography

[1] Erik R. Altman, David Kaeli, and Yaron Sheffer. “Welcome to the Opportunities of Binary Translation”. In: *Computer* 33.3 (Mar. 2000), pp. 40–45. ISSN: 0018-9162.

[2] *AMD SimNow(TM) Simulator 4.6.1: User’s Manual*. 2.14. Advanced Micro Devices, Inc. Nov. 2010.

[3] David Anderson. *White Paper: Red Hat Crash Utility*. `http://people.redhat.com/anderson/crash_whitepaper/`. Accessed: 2014-01-29.

[4] Sina Bahram et al. “DKSM: Subverting Virtual Machine Introspection for Fun and Profit”. In: *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*. IEEE SRDS. New Delhi, India: IEEE Computer Society Press, Oct. 2010, pp. 82–91. ISBN: 978-0-7695-4250-8.

[5] Fabrice Bellard. “QEMU, a Fast and Portable Dynamic Translator”. In: *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference*. USENIX ATC. Anaheim, CA, USA: USENIX Association, Apr. 2005, pp. 41–46.

[6] Nathan L. Binkert et al. “The M5 Simulator: Modeling Networked Systems”. In: *IEEE MICRO* 26.4 (Aug. 2006), pp. 52–60. ISSN: 0272-1732.

[7] Nathan Binkert et al. “The Gem5 Simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (May 2011), pp. 1–7. ISSN: 0163-5964.

[8] Patrick Bohrer et al. “Mambo — A Full System Simulator for the PowerPC Architecture”. In: *ACM SIGMETRICS Performance Evaluation Review* 31.4 (Mar. 2004), pp. 8–12. ISSN: 0163-5999.

[9] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. 3rd ed. O’Reilly Series. Sebastopol, CA, USA: O’Reilly Media, Nov. 2005. ISBN: 978-0-596-00565-8.

[10] Peter M. Chen and Brian D. Noble. “When Virtual Is Better Than Real”. In: *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. HotOS. Elmau, Germany: IEEE Computer Society Press, May 2001, pp. 133–138. ISBN: 0-7695-1040-X.

[11] Bob Cmelik and David Keppel. “Shade: A Fast Instruction-Set Simulator for Execution Profiling”. In: *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. ACM SIGMETRICS. Nashville, TN, USA: ACM, May 1994, pp. 128–137. ISBN: 0-89791-659-x.

[12] Microsoft Corporation. *Debug Help Library*. http://msdn.microsoft.com/en-us/library/windows/desktop/ms679309(v=vs.85).aspx. Accessed: 2014-01-28.

[13] Microsoft Corporation. *Debug Interface Access SDK*. http://msdn.microsoft.com/en-us/library/x93ctkx8.aspx. Accessed: 2014-01-28.

[14] Jiun-Hung Ding et al. “PQEMU: A Parallel System Emulator Based on QEMU”. In: *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*. ICPADS. Tainan, Taiwan: IEEE Computer Society Press, Dec. 2011, pp. 276–283. ISBN: 978-0-7695-4576-9.

[15] Brendan Dolan-Gavitt. *pdbparse: Open-source parser for Microsoft debug symbols (PDB files)*. https://code.google.com/p/pdbparse/. Accessed: 2013-10-31.

[16] Brendan Dolan-Gavitt. *StreamDescriptions*. (Part of the documentation for pdbparse) http://code.google.com/p/pdbparse/wiki/StreamDescriptions. Accessed: 2014-01-28.

[17] Brendan Dolan-Gavitt et al. “Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection”. In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. IEEE SP. Berkeley/Oakland, CA, USA: IEEE Computer Society Press, May 2011, pp. 297–312. ISBN: 978-1-4577-0147-4.

[18] *DWARF Debugging Information Format*. Version 4. DWARF Debugging Information Format Committee. June 2010.

[19] Free Software Foundation, Inc. *DGB Wiki: Internals Breakpoint-Handling*. https://sourceware.org/gdb/wiki/InternalsBreakpoint-Handling. (Accessed: 2014-02-03). Aug. 2013.

[20] Free Software Foundation, Inc. *GDB: The GNU Project Debugger*. www.gnu.org/software/gdb/. Accessed: 2014-01-28.

[21] Tal Garfinkel and Mendel Rosenblum. “A Virtual Machine Introspection Based Architecture for Intrusion Detection”. In: *Proceedings of the Network and Distributed System Security Symposium*. NDSS. San Diego, CA, USA: Internet Society, Feb. 2003. ISBN: 1-891562-16-9.

[22] Emilien Girault. *Volatilitux: Physical memory analysis of Linux systems*. www.segmentationfault.fr/projets/volatilitux-physical-memory-analysis-linux-systems/. Accessed: 2014-01-18.

[23] Alexander Graf. *QEMU's recompilation engine*. http://chemnitzer.linux-tage.de/2012/vortraege/media/videos/v5so1200.mp4. (Accessed 2014-02-27) Recording of Lecture at Chemnitzer Linux-Tage. Mar. 2012.

[24] Silicon Graphics, Sun Microsystems, and David Anderson. *libdwarf*. `www.prevanders.net/dwarf.html`. Accessed: 2013-10-31.

[25] Thorsten Gröninger. "On Statistical Properties of Duplicate Memory Pages". `http://os.ibds.kit.edu/`. Diploma Thesis. System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, Oct. 2013.

[26] R. Nigel Horspool and Nenad Marovac. "An approach to the problem of detranslation of computer programs". In: *The Computer Journal* 23.3 (Aug. 1980), pp. 223–229. ISSN: 0010-4620.

[27] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 3 (3A, 3B & 3C): System Programming Guide. Intel Corporation. Sept. 2013.

[28] Jakub Jelínek. "Improving debug info for optimized away parameters". In: *Proceedings of the GCC Developers' Summit*. GCC Summit. Ottawa, Canada, Oct. 2010, pp. 55–62.

[29] Mark S. Johnson. "Some Requirements for Architectural Support of Software Debugging". In: *Proceedings of the first International Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM ASPLOS. Palo Alto, CA, USA: ACM, Mar. 1982, pp. 140–148. ISBN: 0-89791-066-4.

[30] Nick L. Petroni Jr. et al. "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory". In: *Digital Investigation* 3.4 (Dec. 2006), pp. 197–210. ISSN: 1742-2876.

[31] Michael A. Laurenzano et al. "PEBIL: Efficient Static Binary Instrumentation for Linux". In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE ISPASS. White Plains, NY, USA: IEEE Computer Society Press, Mar. 2010, pp. 175–183. ISBN: 978-1-4244-6022-9.

[32] Kevin P. Lawton. "Bochs: A Portable PC Emulator for Unix/X". In: *Linux Journal* 29es (Sept. 1996). ISSN: 1075-3583.

[33] Jun-Pyo Lee et al. "Aggressive Function Splitting for Partial Inlining". In: *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures*. IEEE INTERACT. San Antonio, TX, USA: IEEE Computer Society Press, Feb. 2011, pp. 80–86. ISBN: 978-1-4577-0834-3.

[34] John R. Levine. *Linkers and Loaders*. 1st ed. The Morgan Kaufmann Series in Software Engineering and Programming. San Francisco, CA, USA: Academic Press, July 1999. ISBN: 978-1-558-60496-4.

[35] Mark A. Lindner. *libconfig – C/C++ Configuration File Library*. `www.hyperrealm.com/libconfig/`. Accessed: 2014-02-09.

[36] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM SIGPLAN PLDI. Chicago, IL, USA: ACM, June 2005, pp. 190–200. ISBN: 1-59593-056-6.

[37] Peter S. Magnusson and Bengt Werner. “Efficient Memory Simulation in SimICS”. In: *Proceedings of the 28th Annual Simulation Symposium*. ANSS. Phoenix, AZ, USA: IEEE Computer Society Press, Apr. 1995, pp. 62–73. ISBN: 0-8186-7091-6.

[38] Peter S. Magnusson et al. “Simics: A Full System Simulation Platform”. In: *Computer* 35.2 (Feb. 2002), pp. 50–58. ISSN: 0018-9162.

[39] Milo M. K. Martin et al. “Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset”. In: *ACM SIGARCH Computer Architecture News* 33.4 (Sept. 2005), pp. 92–99. ISSN: 0163-5964.

[40] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. 1st ed. Wrox Programmer to Programmer. Hoboken, NJ, USA: Wiley, Oct. 2008. ISBN: 978-0-470-34343-2.

[41] Julia Menapace, Jim Kingdon, and David MacKenzie. *The "stabs" debug format (Revision: 2.125)*. `http://docs.freebsd.org/info/stabs/stabs.pdf`. Accessed: 2014-01-28.

[42] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM SIGPLAN PLDI. San Diego, CA, USA: ACM, June 2007, pp. 89–100. ISBN: 978-1-59593-633-2.

[43] Avadh Patel, Furat Afram, and Kanad Ghose. “MARSS-x86: A QEMU-Based Micro-Architectural and Systems Simulator for x86 Multicore Processors”. In: *Proceedings of the 1st International QEMU Users’ Forum*. International QEMU Users’ Forum. Grenoble, France, Mar. 2011, pp. 29–30.

[44] Avadh Patel et al. “MARSS: A Full System Simulator for Multicore x86 CPUs”. In: *Proceedings of the 48th Design Automation Conference*. DAC. San Diego, CA, USA: ACM, June 2011, pp. 1050–1055. ISBN: 978-1-4503-0636-2.

[45] Avadh Patel et al. *MARSSx86 - Micro-ARchitectural and System Simulator for x86-based Systems*. `www.marss86.org`. (Accessed: 2014-02-12). Feb. 2013.

[46] *QEMU Emulator User Documentation*. QEMU 0.12.0 release. QEMU.org. Jan. 2010.

[47] Mendel Rosenblum et al. “Using the SimOS Machine Simulator to Study Complex Computer Systems”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 7.1 (Jan. 1997), pp. 78–103. ISSN: 1049-3301.

[48] Christian Schneider, Jonas Pfoh, and Claudia Eckert. “A Universal Semantic Bridge for Virtual Machine Introspection”. In: *Proceedings of the 7th International Conference on Information Systems Security*. ICISS. Kolkata, India: Springer-Verlag, Dec. 2011, pp. 370–373. ISBN: 978-3-642-25559-5.

[49] Sven B. Schreiber. *Undocumented Windows 2000 Secrets: A Programmer’s Cookbook*. 1st ed. Indianapolis, IN, USA: Addison-Wesley Professional, May 2001. ISBN: 978-0-201-72187-4.

[50] Stan Shebs. “GDB Tracepoints, Redux”. In: *Proceedings of the GCC Developers' Summit*. GCC Summit. Montréal, Canada, June 2009, pp. 105–112.

[51] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 7th ed. Hoboken, NJ, USA: Wiley, Jan. 2005. ISBN: 0-471-69466-5.

[52] Amitabh Srivastava and Alan Eustace. “ATOM: A System for Building Customized Program Analysis Tools”. In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*. ACM SIGPLAN PLDI. Orlando, FL, USA: ACM, June 1994, pp. 196–205. ISBN: 0-89791-662-x.

[53] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. For GCC version 4.8.2. GNU Manual. Boston, MA, USA: GNU Press.

[54] Richard M. Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*. Tenth Edition for DGB version 7.7.50.20140128-cvs. GNU Manual. Boston, MA, USA: Free Software Foundation, Inc. ISBN: 978-0-9831592-3-0.

[55] Volatile Systems. *The Volatility Framework: Volatile memory artifact extraction utility framework*. `www.volatilesystems.com/default/volatility/`. Accessed: 2013-10-31.

[56] Ariel Tamches and Barton P. Miller. “Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels”. In: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*. OSDI. New Orleans, LA, USA: USENIX Association, Feb. 1999, pp. 117–130. ISBN: 1-880446-39-1.

[57] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Version 1.2. Tool Interface Standard Committee. May 1995.

[58] *Tool Interface Standard (TIS) Formats Specification for Windows*. 1.0. Tool Interface Standard Committee. Feb. 1993.

[59] Emmett Witchel and Mendel Rosenblum. “Embra: Fast and Flexible Machine Simulation”. In: *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS. Philadelphia, PA, USA: ACM, May 1996, pp. 68–79. ISBN: 0-89791-793-6.

[60] Jürgen Wolf. *C von A bis Z: Das umfassende Handbuch*. 2nd ed. Galileo Computing. Bonn, Germany: Galileo Press, Jan. 2006. ISBN: 3-89842-643-2.

[61] Matt T. Yourst. “PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator”. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE ISPASS. San Jose , CA, USA: IEEE Computer Society Press, Apr. 2007, pp. 23–34. ISBN: 1-4244-1081-9.