Lightweight Memory Tracing

Mathias Payer
ETH Zurich

Enrico Kravina
ETH Zurich

Thomas R. Gross
ETH Zurich

Abstract

Memory tracing (executing additional code for every memory access of a program) is a powerful technique with many applications, e.g., debugging, taint checking, or tracking dataflow. Current approaches are limited: software-only memory tracing incurs high performance overhead (e.g., for Libdft up to 10x) because every single memory access of the application is checked by additional code that is not part of the original application and hardware is limited to a small set of watched locations.

This paper introduces *memTrace*, a lightweight memory tracing technique that builds on dynamic on-the-fly cross-ISA binary translation of 32-bit code to 64-bit code. Our software-only approach enables memory tracing for unmodified, binary-only x86 applications using the x64 extension that is available in current CPUs; no OS extensions or special hardware is required. The additional registers in x64 and the wider memory addressing enable a low-overhead tracing infrastructure that is protected from the application code (i.e., uses disjunct registers and memory regions). MemTrace handles multi-threaded applications. Two case studies discuss a framework for unlimited read and write watchpoints and an allocation-based memory checker similar in functionality to memgrind.

The performance evaluation of memTrace shows that the time overhead is between 1.3x and 3.1x for the SPEC CPU2006 benchmarks, with a geometric mean of 1.97x.

1 Introduction

Analyzing memory accesses in large applications is a hard problem due to limitations of the current tracing infrastructure and hardware. Dynamic program instrumentation that naively instruments every memory access results in high execution overhead (20x for Valgrind's memcheck [18], up to 10x for libdft [14], up to 21.1x for compression for PTT [9], and up to 40x for taintcheck [19]), and the execution overhead makes it often impossible to execute large instrumented applications up to the point where a specific bug is triggered. Hardware watchpoints are limited to a small set of memory locations but allow tracing at native performance.

Memory tracing allows the execution of *memlets* for every memory access of the instrumented application. Memlets are code sequences that are woven into the executed application code. These memlets can execute additional code for each memory access depending on: (i) the data value that is read or written, (ii) the address that is read from or written to, or (iii) the state associated with the address that is read or written (the tracing infrastructure may provide additional state – a *shadow value* – for every memory location that is used in an application). Memory tracing is lightweight if the overall performance overhead added through the memlets is low. Memlets can use the state and the value of each memory location to implement high-level functionality like (unlimited) watchpoints, dataflow tracking, or taint checking.

This paper presents *memTrace*, a framework for lightweight memory tracing for single-threaded and multi-threaded 32-bit applications. MemTrace combines an API to set and check shadow values for every byte used in the application with an interface to implement different user-defined memlets. We present two example memlets that (i) support an unlimited number of memory watchpoints and (ii) enforce explicit safety regions around every memory allocation for C/C++ applications to find memory corruption bugs like buffer overwrites and buffer underwrites, and these memlets handle arbitrary unmodified 32-bit binary applications.

Current memory tracing systems use software binary translation to instrument all memory accesses of an application with a pre-determined set of instructions (i.e., current systems do not support user-configurable memlets). Some systems reuse "unused" registers (e.g., minemu [4] uses the SSE registers and therefore only supports applications that do not use SSE instructions, LIFT [23] uses x64 registers) while other systems (e.g., PIN [15], or Valgrind [18]) reallocate registers during the binary translation process. Unused registers speed up memory tracing because the memlets and the memory checks use these registers, and no spill code is needed.

All recent Intel and AMD x86 CPUs are x64 capable, on the other hand most applications are based on the 32-bit x86 ISA (e.g., the recommended Ubuntu end-user image uses only 32-bit applications, and all Windows and MacOS operating system images exist in a 32-bit and a 64-bit version). A drawback of x64 is the increased memory usage due to the 64-bit pointer width and the larger page tables. Most applications fit well into a 32-bit memory space. MemTrace enables lightweight memory tracing for these common x86 applications and uses the available features of the already dominant x64 hardware. The combination of free registers to implement the lookup checks and a data structure that supports fast and efficient lookup for individual memory locations is key to low execution overhead.

MemTrace uses cross-ISA translation for 32-bit applications to a 64-bit ISA to offer both a wider address space and additional registers to user-defined memlets. The memTrace prototype implementation leverages the x64[1] ISA to implement efficient memory tracing for unmodified x86 applications. The x86 code is dynamically translated to x64 code. The x64 ISA is the 64-bit extension of x86. Most instructions are available in both ISAs and can be translated easily. The cross-ISA translation provides 8 additional registers that can be used for the memlets. A shadow memory area above the 4GB limit of the 32-bit x86 application (i.e., application code uses only 32-bit pointers and is therefore unable to interfere with the shadow memory area) is used to store the data used by the memlets. Our prototype implementation of memTrace supports all x86 instructions, including all FPU and SSE extensions.

The flexible implementation of memlets combined with shadow data enables additional fine-grained operations that build on top of memory tracing like dataflow tracking or taint checking. The memlets update the data or taint information for each memory location and check the integrity of the data upon every memory access.

A key observation of Greathouse et al. [12] is that a fast memory tracing framework needs some form of additional hardware extensions to achieve low overhead. This paper shows that low overhead memory tracing can be achieved in software by using additional hardware resources (more registers and a wider address space) that are available through dynamic cross-ISA translation. The memTrace memory tracing technique offers new opportunities for debugging, dataflow tracking, or other user-defined memlets that evaluate fine-grained memory access.

The memTrace prototype implementation supports arbitrary applications like the OpenOffice office suite or the Apache webserver. A performance evaluation of the memTrace prototype implementation for x64 Linux kernels with the SPEC CPU2006 benchmarks shows low overhead with a geometric mean of 1.97x. The contributions of this paper are as follows:

1. The architecture of *memTrace*, a lightweight memory tracing technique for binary-only 32-bit applications that supports user-defined memlets and leverages cross-ISA translation.
2. A case study that shows two memlets: one that supports unlimited watchpoints and a second one that checks an application for memory allocation errors (allocation over-writes and under-writes).
3. An evaluation and discussion of a prototype implementation of the memory tracing technique for x86/x64 and the corresponding memlets.

The rest of the paper is organized as follows: Section 2 lists requirements for lightweight memory tracing; Section 3 describes cross ISA binary translation; Section 4 shows two case studies that use memory tracing; Section 5 presents the memTrace implementation; Section 7 discusses related work; and Section 8 concludes.

2 Requirements for lightweight memory tracing

MemTrace is a technique for lightweight memory tracing that builds on dynamic cross-ISA binary translation. Dynamic binary translation keeps the overhead low and cross-ISA translation from 32-bit to 64-bit enables the memlets to access a broader memory space than the original ISA permits. This paper discusses 32-bit programs running on a 64-bit ISA. Other combinations work analogously, e.g., 16-bit code running on a 32-bit ISA, as long as the address space of the target ISA is a super-set of the source ISA. A lightweight memory tracing technique must fulfill the following requirements:

Unchanged application address space: the 32-bit application has access to the full 4GB memory space. The larger target address space allows memTrace to hide the binary translation framework and all the data structures needed by the memlets from the application. Neither the binary translator nor the memlets store any internal state in the application memory space. The memlets may change application memory values as part of their functionality. This requirement ensures that the binary translator does not interfere with the original memory layout of the application and, e.g., the placement of shared libraries.

Unmodified execution: the translated application follows the same control flow pattern as the original application. The application uses the original return addresses on the stack, the original function pointers, and the original targets for indirect jumps. The translated code executes additional lookups in a mapping table to transparently map from translated to untranslated code targets. This requirement ensures that the application can use original addresses, e.g., as control flow targets.

Full isolation: the application has no access to data of the binary translator or to data of the memlets. The translated application cannot access any data above the original application segment (due to the restriction of 32-bit pointers). This requirement ensures that the application cannot modify any internal data.

Flexible memlets: the memory tracing technique enables the implementation of flexible memlets that use shadow memory or registers as state. MemTrace allows the implementation of any memlet that needs one or more bytes of state for each byte that the application uses. The memlets can use additional available free registers in the target ISA.

Low overhead: the overall overhead of the memory tracing technique must be low, and the other requirements must not preclude a fast implementation.

The technique for lightweight memory tracing presented in this paper fulfills the criteria above.

[1] Multiple different names are used for the 64-bit extension of x86: x64, EM64T, AMD64, IA-32e, and x86-64. This paper uses x64.

3 Cross-ISA binary translation

Cross-ISA binary translation takes a program written in a source ISA and executes the program on a different target ISA. Multiple reasons for cross-ISA translation exist, e.g., program portability, or additional resources in the target ISA. Depending on the differences between the two ISAs the translation is non-trivial.

This paper discusses two different architectures that are both extended from 32-bit to 64-bit, namely the Intel x86 platform and the ARM platform. The first example covers the x86 ISA. The x86 ISA evolved over more than 20 years and was extended multiple times. 32-bit x86 and the 64-bit extension x64 are closely related. x64 widens the registers and the address size to 64-bit, adds 8 general purpose registers, and introduces new instructions. Some instructions are removed as well: 16 one-byte x86 instructions are replaced and reused as prefixes for the new x64 instructions. The original 16 instructions are no longer available on x64 and must be emulated using longer instructions. Additional changes include (i) the limitation of segmentation which makes binary translation for x64 harder [27] and (ii) the way system calls are executed.

A second example is binary translation for the ARM platform. The ARMv8-A architecture supports two ISAs: the AArch64 ISA is a 64-bit extension of the 32-bit AArch32 ISA. Similar to the x64 extension of x86 AArch64 supports a wider address space and a wider register file. The prototype implementation focuses on x86/x64 but the design of the memTrace technique is applicable to AArch64/AArch32 as well because our technique relies on a wider address space and similar instructions between the two ISAs. A notable difference between x86 and ARM is that the instruction pointer (EIP) cannot be accessed directly on x86 while it is a regular register on ARM. The binary translator modifies the EIP to execute translated code from the code cache but emulates all instructions that indirectly use the EIP to keep up the illusion of an unmodified application (e.g., `call foo` is translated into `push orig_eip; jmp transl_foo`. In contrast to x86, an ARM implementation must emulate all instructions that use the program instruction counter directly as well.

There are two problems that must be solved for binary translation for 32-bit x86 programs: register pressure and location of internal data structures of the binary translator. Register allocation on x86 is a hard problem [1, 2, 29] and register reallocation in a binary translator without type information and control-flow information is even more complex. Translating 32-bit x86 applications to x64 code solves the register pressure problem. The 8 additional registers are used by both the dynamic binary translator to implement the translation process and the memlets to implement the memory tracing. The translated application uses the unchanged original registers except for the program counter. Same-ISA binary translators modify the original memory address space of an application by placing internal data structures somewhere into the existing memory space. Cross-ISA translation from x86 to x64 enables a wider address space. Consequently, the translated application uses the low 4GB of memory and the binary translator and the memlets place their data in the upper memory areas. The translated application keeps using 32-bit pointers and cannot access the memory of the binary translator.

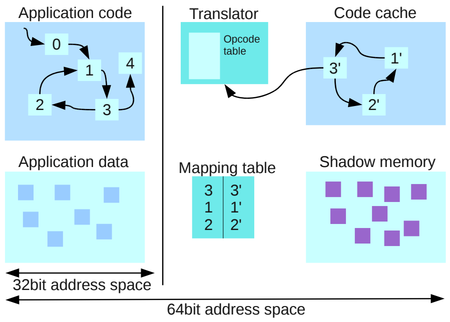


Figure 1: Binary translator runtime layout. Basic blocks are translated and placed in the code cache using opcode tables. The mapping table maps addresses in the program to translated addresses. Trampolines invoke the translator for untranslated basic blocks.

To summarize, the advantages of cross-ISA binary translation are: (i) additional registers available for the instrumentation, (ii) memory separation as translated x86 code cannot access the code of the translator, and (iii) full encapsulation of the translated application.

A possible disadvantage is that some hardware features like segmentation are limited. Fortunately segmentation is not used in user-space applications except for thread local data. Segmentation for thread local data is still supported on x64.

3.1 Dynamic binary translation

Dynamic binary translation instruments a user-space application on the fly. Figure 1 shows the design of the dynamic binary translator and the memory layout. The translator compiles individual basic blocks of the original x86 application on demand and places the translated code in a code cache. Translated control flow transfers use the mapping table to translate targets in the original application to targets in the code cache. Untranslated target fall back to the translator. All executed code is either a part of the translator or of the generated code.

Instructions are translated using a table-based translation scheme as described in libdetox [21]. Most instructions are copied verbatim. For cross-ISA translation some instructions must be adapted due to different memory encodings or addressing schemes, other instructions are emulated by the translation layer. In addition, all instructions that alter control flow (e.g., jump instructions, call instructions, return instructions, system calls, or interrupts) are adapted so that the binary translator keeps control of the translated application.

3.2 Memory layout

The x64 ISA uses 64-bit wide pointers (whereas the physical memory bus is up to 48bit wide). The binary translator maps the original application to the low 4GB. The binary translator library, the code cache, the mapping table, and all shadow memory are placed above the 4GB limit of the translated application.

The translated application still uses 32-bit pointers and 32-bit registers and has therefore no access to any data of the binary translator. This enables *hardware enforced protection* of the internal data from the translated application as the application is not able to generate a memory access to that region due to the 32-bit wide pointers used in the source ISA. In addition, the application has exclusive access to the original 32-bit address space; the binary translator keeps all data in higher memory areas.

3.3 MemTrace design summary

All state for the memlets and the internal data for the binary translator are stored in the area of the 64-bit address space above the first 4GB. The wider address space of a 64-bit ISA like x64 in comparison to a 32-bit address space allows the binary translator to place the shadow memory data structure and all binary translator data structures into an address area that is not accessible from the original application. The translated application uses the low 4 GB of the 64-bit address space that overlaps with the complete 4 GB address space of a 32-bit application. The binary translator is completely concealed; translated code is put in a code cache and every control flow transfer uses a mapping table to map the original target in the application memory space to the translated target in the binary translator space. The application is fully isolated from the binary translator: all pointers in the application domain are 32-bit; the application has no access to any data of the binary translator. The evaluation of the prototype implementation shows low performance overhead with a geometric mean of 1.97x for the SPEC CPU2006 benchmarks.

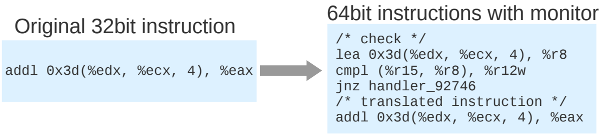
4 Memory tracing case studies

This section presents two case studies that use the lightweight memory tracing technique. The first case study designs a memlet for unlimited watchpoints. The memlet for unlimited watchpoints supports both read and write watchpoints and can be used to overcome the hardware limitation of 4 write watchpoints on current x86 platforms.

The second case study implements a memory allocation checker. Upon every allocation in a C or C++ program the memory checker adds additional safe zones around the allocated memory region. Any out-of-bounds reads and writes are detected and stop the program.

4.1 Case study: a memlet for unlimited watchpoints

Watchpoints are used to debug applications and enable the inspection of specific memory addresses. Read watchpoints are triggered whenever the location is read and write watchpoints are triggered whenever the location is written. For example, if a certain address is read or written by a bug in the application then a watchpoint can be used to find the code location and context where that read or write access is executed. x86 supports up to 4 (up to 8 byte wide) hardware watchpoints that can be set using debug registers. For many use cases 4 watchpoints are not enough as a wider memory region must be protected to find a specific bug.



```
addl 0x3d(%edx, %ecx, 4), %eax
```

```
/* check */
lea 0x3d(%edx, %ecx, 4), %r8
cmpl (%r15, %r8), %r12w
jnz handler_92746
/* translated instruction */
addl 0x3d(%edx, %ecx, 4), %eax
```

Figure 2: Translation of a memory accessing instruction.

The lightweight memory tracing technique facilitates the design of a simple watchpoint memlet that implements unlimited read and write watchpoints with constant overhead. The overhead is constant for every memory access and does not increase with the number of watchpoints.

The watchpoint memlet uses a shadow memory segment of the same size as the original application. The shadow memory is mapped with a 4GB offset (i.e., the address `0xdeadbeef` is shadowed at `0x1deadbeef`). Every byte in shadow memory is either 0 (if no watchpoint should be triggered) or non-0 if either a read or write watchpoint is set. Figure 2 shows the translation of a sample instruction that reads a memory address. The instruction is translated to 64-bit by expanding all pointers in the instruction. MemTrace adds the memlet before the memory-accessing instruction and checks if the shadow data is 0. The register `%r15` holds the constant offset `0x100000000`, `%r12` keeps the value 0, and `%r13` is used to store the watchpoint information.

The memlet is optimized for fast execution: the instruction cache (i-cache) pressure is reduced by using shorter instruction encodings for memlets and moving the watchpoint handler (the cold path) out into a trampoline. The memlet uses two registers (`%r12` and `%r15`) to store constants. Each replacement of a constant with a register saves 8 bytes in the instruction length. In addition, the translator generates a cold path trampoline for each instruction that accesses memory. The trampoline stores the context (i.e., original IP of the instruction that triggered the watchpoint) and transfers control to the general watchpoint handler.

An interesting feature of the shadow memory segments is that unaligned multi-byte memory accesses are supported. If an instruction accesses a multi-byte value then the shadow bytes of all bytes are combined. The memlet checks for non-0 and detects with a single check if a watchpoint is set for at least a single byte of the multi-byte access.

The watchpoints can either be used by a debugging script/program or can be used as regular watchpoints in GDB. GDB allows remote stubs as backends with the standard GDB frontend using a remote serial protocol [10]. The backend implements a simple protocol to, e.g., read registers, set breakpoints, and to set watchpoints. The remote stub starts the application under the control of the lightweight memory tracing prototype implementation. Watchpoints are forwarded from the GDB frontend and activated using the watchpoint memlets in translated code.

4.2 Case study: safe heap memory allocator

Ptmalloc2 [11], the standard allocator for C and C++ is an in-place memory allocator that stores information about each allocated memory block before and after the block. This information may be overwritten by buffer overflows or random memory writes. Such bugs are hard to find because memory corruption bugs might only cause a segmentation fault when the block is reused the next time.

This case study uses the watchpoint memlet to set watchpoints before and after every allocated memory block. Calls to the memory allocator are intercepted by the binary translation framework and new watchpoints are added dynamically. If a block is collected (freed) then the watchpoints are removed.

If a bug in the application writes to a watchpoint or reads a watchpoint (i.e. the application accesses an illegal memory region) then the application is either terminated with an information message or a debugger is attached dynamically so that a programmer can analyze the problem.

5 Implementation

The prototype implementation of memTrace extends the libdetox [21] binary translation platform. The libdetox platform is a table-based x86 to x86 binary translator. Our prototype implementation extends the translator with a cross-ISA translation module that transforms x86 instructions to equivalent x64 instructions. The complete prototype implementation is released as open source.

The prototype implementation maps the 32-bit version of the standard loader `ld.so` into the 32-bit address space and prepares the application stack with the correct parameters that `ld.so` expects for the initialization of a 32-bit application. Next the binary translator starts translation and execution of the loader code which loads and initializes all needed shared libraries and starts the execution of the application.

The following sections discuss the translation of individual instructions, present how the memory layout of a translated application looks, and focus on specific translation details.

5.1 Instruction translation

Due to the similarity of the two ISAs the encoding of most instructions is similar as well, the translation is straight-forward and follows the concept of other table-based translators. For most instructions the available encodings on x64 are a super-set of the available encodings of x84. The binary translator uses linked instruction tables to decode the current instruction. If the instruction accesses memory then the pointers are zero expanded to 64-bit memory addresses and the memlet is emitted before the translated instruction. The binary translator uses one of the following translation schemes for each instruction:

Emulation: instructions that are not available on x64 (e.g., `pusha`, or `popa`) are replaced by a sequence of instructions that emulates the removed instruction transparently.

Exception: instructions that are no longer used (e.g., `aaa`, or `aad`) raise an exception. The binary translator fails gracefully and prints an error message. An emulation of these instructions can be added if needed.

Encoding: some instructions are encoded differently on x64 (e.g., `inc`, or `dec`). These instructions are replaced during the translation process.

Addressing mode: x64 uses an instruction pointer relative addressing mode instead of an absolute addressing mode. Absolute references are translated dynamically to absolute addresses during code generation.

Different semantics: some instructions change their semantics (e.g., `push`, or `pop`) and operate on quadwords on x64. These instructions are translated to operate on doublewords during the translation.

Rep prefix: the handling of the rep prefix changes for x64. String operations (e.g., `rep stosb`) that use the rep prefix are translated to loops during the translation process.

On x64 segment-based addressing is restricted compared to x86. Current user-space x86 applications use segmenting only for thread local storage in current applications. The x64 ISA supports segmentation for thread local storage and the prototype implementation support 32-bit thread local storage in a 64-bit environment.

5.2 Shadow memory

The 64-bit address space enables the implementation to use address regions that cannot be encoded using 32-bit memory pointers. The application uses the low 4GB of the 64-bit address space and no data in that region is changed through the binary translator (the data may be changed as a function of the memlets). Figure 3 shows the memory layout of a running application under the control of memTrace.

The next 4GB are used as shadow memory of the application memory at offset `0x100000000`. The memlets store information about the corresponding memory addresses of the application in the shadow memory. The shadow memory regions are mapped at the same time when the application memory is mapped. Virtual memory allocates physical pages only if the page is accessed by a memlet (e.g., code regions are not accessed by our memlets and the physical pages are therefore not allocated). An upper bound for the memory consumption for the shadow memory is 1x the

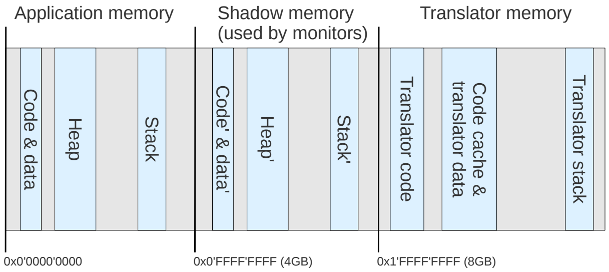


Figure 3: Memory layout of the translated application.

memory that the application uses. Memlets may use multiple shadow memory regions to store additional information (i.e., at offsets `0x200000000`, `0x300000000`, and so on).

5.3 Register allocation

The x64 ISA offers 8 additional general registers (`%r8` to `%r15`) that can be used by the lightweight memory tracing technique. The binary translator component is compiled for the x64 ISA and uses all available registers during the translation of x86 code. The transition between translator and translated application code saves and restores all general purpose registers. The memlets are native x64 code and can use the 8 additional x64 registers during the execution of the translated code.

Both memlets discussed here use registers `%r8` and `%r9` as temporary scratch registers. The memory watchpoint memlet uses three registers `%r10`, `%r11`, and `%r13` to track usage of the `eflags` register. Saving and restoring the `eflags` register before and after the execution of a memlet adds overhead, therefore reducing the number of save and restore operations is important.

The register `%r12` holds the constant `0x0` and the register `%r15` holds the constant offset to the shadow memory (`0x100000000`). Using registers to hold constants instead of encoding constants in the instruction itself saves 8 bytes per used constant in the emitted code. If needed, these registers may be used for other purposes.

5.4 String instructions

String instructions use the `rep` prefix to repeat a single instruction n times. String instructions access multiple memory locations in a sequence of incrementing or decrementing addresses.

MemTrace replaces string instructions with a short loop that first checks the source address and the destination address, executes a single instruction with the current parameters, and increments or decrements the source and target registers.

5.5 System calls, signals, and threads

An x86 application requests system calls either using interrupts (`int $0x80`) or using the `sysenter` instruction. On x64 the application uses the `syscall` instruction. The mapping between system call and system call number is different between x86 and x64. The parameters of individual system calls can change as well (i.e., 64-bit wide addresses instead of 32-bit wide addresses).

MemTrace uses a mapping table to map between x86 and x64 system calls. Most system calls can be mapped easily. For system calls that access memory, pointers are dynamically extended to 64-bit and returned pointers from the kernel are truncated to 32-bit. All memory management system calls (`mmap`, `munmap`, `mremap`, and `brk`) are redirected to a special handler function that checks and adapts the specified parameters and manages the shadow memory as well.

Signals are handled in the binary translator as well. The binary translator intercepts all system calls that install signals and replaces the signal handlers with its own internal signal handler. This signal handler then handles the switch to the application stack if the signal was caught while in the translator and executes the corresponding translated application signal handler.

The cross-ISA binary translator supports Linux pthreads by translating thread-related system calls of the application into the necessary 64-bit system calls. Thread support is a difficult problem for memory tracing due to possible synchronization issues. Two threads may concurrently modify the same memory address and the corresponding memlets may therefore access the same shadow value. As long as the application synchronizes access to the memory location the access to the shadow value is implicitly synchronized as well. If the original program has a data-race then the memlets must synchronize concurrent writes, e.g., by using regional locks. Simply adding a `lock` prefix to the original memory access is not enough as the memlet will access a second memory location. Adding explicit locks for each memory access adds high overhead and is currently not implemented.

The accesses of the memlets to the shadow table follow the same pattern as the memory accesses in the original application. If the application locks the memory region for a specific thread then the corresponding shadow memory region is implicitly locked as well. No other application thread can access the original memory region, therefore no memlet of another thread will access the implicitly locked shadow memory region. This implicit locking approach only works if the application has no data races between threads. User-defined memlets that analyze inter-thread behavior, e.g., to check for data races, must lock the shadow memory themselves.

5.6 Flag tracking optimizations

This section presents two optimizations for the binary translator that help lowering the overhead for memory tracing. The first optimization tracks the usage of the `eflags` register and allows the memlets to change the `eflags` register if the `eflags` register is not used between instructions that affect the flags. The second optimization stores the operands of the relevant arithmetic instruction in two free registers and reexecutes the instruction with a bogus target.

A big advantage of the cross-ISA translation is that

```
addl %ebx, %eax # sets flags, unused
subl %ecx, %eax # sets flags, used
movl (0xdeadbeef), %ebx # mem. read
movl %eax, (0xdeadbeef) # mem. write
jnz next_block  # uses flag
```

Listing 1: Example of a basic block in an application

memTrace can use additional free registers without the need for register reallocation. Unfortunately one register is shared between the binary translator, the memlets, and the translated application code: the `eflags` register. The situation is worsened by the fact that access to this register is very slow when using `pushf` and `popf` instructions. Instead of using `pushf` and `popf` instructions we use a short sequence of instructions (`lahf` and `seto` to save and `addb` and `sahf` to restore) to handle the `eflags` register.

On x86 all arithmetic instructions, the "compare" instruction, and the "test" instructions set the `eflags` register. But the `eflags` register is only read after a subset of the instructions. Table-based binary translators do not build an intermediate representation (IR) which makes `eflags`-usage tracking more complicated. MemTrace uses a triple-pass approach to track usage of the `eflags` register in each basic block. The first two passes decode all instructions and analyze which instructions use the `eflags` register. The third pass emits translated instructions (including the memlets which change the application-set `eflags` register) but inserts code that saves the `eflags` register only when necessary.

The second optimization saves the operands of the `eflags`-relevant arithmetic instruction in two x64 registers (`%r10`, and `%r11`). In front of the instruction that reads the `eflags` register the arithmetic instruction is executed again (with the saved operands) to reproduce the state of the `eflags` register. This optimization reduces the overhead of saving and restoring the `eflags` register in tight loops.

Listing 1 shows an example of a basic block. The first two instructions set the `eflags` registers, but only the result of the `subl` instruction is used. The two `movl` instructions execute memory accesses and the memlets in the instrumented code overwrite the status of the `subl` instruction. MemTrace restores the status of the `eflags` register of the last arithmetic instruction before the `jnz` instruction.

6 Evaluation

The prototype implementation is stable and runs applications like, e.g., the parsec benchmarks, OpenOffice, gedit, and the complete set of SPEC CPU2006 benchmarks. The evaluation uses the SPEC CPU2006 benchmarks to evaluate the performance of the memTrace prototype implementation, including two different user-defined memlets.

This evaluation uses all SPEC CPU2006 benchmarks except 481.wrf which no longer compiles on modern systems. This is not a limitation of our prototype implementation but a limitation of the SPEC CPU2006 benchmarks.

All benchmarks are executed on a 64-bit version of Ubuntu 12.04. The machine uses an Intel Core i7-2640M CPU with 2 cores at 2.80 GHz with 4 GB of memory. The benchmarks are compiled using gcc version 4.6.3 and use the glibc version 2.15. The benchmarks are compiled for 32-bit.

6.1 SPEC CPU2006

This section evaluates the performance of memTrace, our prototype implementation, using the SPEC CPU2006 version 1.0.1 benchmarks using the flags `-O3 -m32`. We evaluate different configurations of memTrace to show the overall overhead and relative performance changes for individual optimizations. The evaluations use the `runspec` script to produce reproducible runs with 3 iterations.

We perform the measurements on both the *reference* dataset and on the *test* dataset. The test dataset is used to evaluate the overhead for short running programs while the reference dataset shows the overhead for long running benchmarks. The following configurations are used:

NAT: A native configuration that runs without binary translation or memory tracing.

ID: The benchmarks execute with binary translation.

EFL: This configuration measures the overhead for storing and restoring the `eflags` register for memory tracing. Code that saves and restores the `eflags` register is added before as if memory tracing is executed but no memlets are added. All optimizations discussed in Section 5.6 are enabled.

MT: This configuration shows the performance of the baseline memory tracing framework. MT extends the EFL extension and measures the impact of reading the shadow memory address for each memory access.

WP: This configuration executes full memory tracing using the watchpoint memlets (without any active watchpoints).

Table 1 shows the overhead of the four different memTrace configurations compared to native execution of the 32-bit binaries. Most benchmarks exhibit moderate overhead for the different memTrace configurations. The overhead is always below 3.11x and for 16 of 28 applications the overhead is below 2x.

The ID configuration measures the overhead for cross-ISA translation. The overhead for cross-ISA binary translation is low, 15% on average with a geometric mean of 17%. The usual culprits 400.perlbench, 403.gcc, 445.gobmk, 458.sjeng, and 453.povray result in an overhead of more than 40% for binary translation due to the high number of indirect control flow transfers. The ID configuration shows that the binary translator is a reasonable baseline to implement memory tracing.

Benchmark	NAT [s]	ID	EFL	MT	WP
400.perlbench	324.00	1.74	2.10	2.60	2.82
401.bzip2	498.00	1.08	1.28	1.91	1.97
403.gcc	305.00	1.40	1.65	2.08	2.20
429.mcf	278.00	1.10	1.10	2.00	2.21
445.gobmp	434.00	1.49	1.85	2.26	2.74
456.hmmer	433.00	1.01	1.43	2.63	2.63
458.sjeng	485.00	1.56	1.99	2.35	2.72
462.libquantum	543.00	1.01	1.03	1.23	1.24
464.h264ref	609.00	1.22	1.42	2.71	2.86
471.omnetpp	308.00	1.37	1.44	1.89	1.94
473.astar	412.00	1.09	1.25	1.60	1.62
483.xalancbmk	252.00	1.90	2.23	2.68	2.99
410.bwaves	405.00	1.01	1.20	1.96	1.97
416.gamess	760.00	1.09	1.57	2.30	2.43
433.milc	441.00	1.00	1.06	1.32	1.34
434.zeusmp	540.00	1.02	1.22	1.69	1.72
435.gromacs	658.00	1.01	1.15	1.43	1.49
436.cactusADM	1120.00	0.99	1.24	2.21	3.11
437.leslie3d	447.00	1.02	1.11	1.44	1.44
444.namd	412.00	1.02	1.23	1.61	1.63
447.dealII	318.00	1.35	1.56	2.08	2.13
450.soplex	298.00	1.07	1.23	1.46	1.51
453.povray	181.00	1.49	2.05	2.62	2.78
454.calculix	696.00	1.03	1.21	1.55	1.59
459.GemsFDTD	503.00	1.04	1.14	1.61	1.66
465.tonto	559.00	1.15	1.35	1.71	1.81
470.lbm	440.00	1.00	1.02	1.13	1.14
482.sphinx3	521.00	1.05	1.23	1.59	1.61
Average	470.71	1.15	1.36	1.90	2.06
Geo. mean	439.54	1.17	1.37	1.86	1.97

Table 1: Performance evaluation using the SPEC CPU 2006 benchmarks (*reference* dataset). NAT shows native execution in seconds, the remaining columns show memTrace configurations relative to NAT.

The EFL configuration measures the performance overhead induced by `eflags` tracking, saving, and restoring needed if additional code is executed for every memory-accessing instruction. No memlet code is executed for this configuration. The average performance overhead for this configuration is 36% and the geometric mean is 37%. Different benchmarks show different increase in the performance overhead. These differences hint at the number of memory-accessing instructions that are executed for each benchmark. If the overhead increases over-proportional, then the benchmark executes more memory accessing instructions than the average benchmark.

The MT configuration extends the EFL configuration by reading the shadow value for each accessed memory location. No additional computation is executed. The performance difference between EFL shows the impact of one additional `mov` instruction per memory-accessing instruction combined with additional cache pressure for accessing twice as many memory locations in hot code regions. Several benchmarks exhibit a performance impact of 2.0x to 2.7x for this configuration. The average overhead is 1.90x with a geometric mean of 1.86x. This configuration shows the overhead for memory tracing without executing any memlets.

Benchmark	NAT [s]	ID	EFL	MT	WP	VAL	VMEM
400.perlbench	3.57	1.67	1.75	1.88	1.95	6.53	err
401.bzip2	5.79	1.10	1.30	2.09	2.16	4.40	16.29
403.gcc	1.00	2.01	2.27	2.74	3.07	11.42	err
429.mcf	1.69	1.15	1.19	2.17	2.27	3.44	10.00
445.gobmk	15.00	1.49	1.85	2.26	2.71	8.07	31.20
456.hmmer	2.51	1.10	1.49	2.37	2.39	5.66	28.53
458.sjeng	3.39	1.53	1.87	2.25	2.58	7.73	31.27
462.libquantum	0.05	1.40	1.56	2.01	2.01	8.30	17.35
464.h264ref	12.30	1.24	1.59	2.60	2.72	4.73	34.47
471.omnetpp	0.31	3.66	4.33	4.90	5.92	18.15	73.25
473.astar	7.93	1.08	1.23	1.54	1.59	2.86	11.92
483.xalancbmk	0.08	4.36	4.52	5.39	5.79	22.93	65.79
410.bwaves	5.13	1.02	1.21	2.03	2.05	5.85	61.79
416.gamess	0.33	1.47	1.79	2.40	2.58	11.23	43.69
433.milc	5.06	1.12	1.34	1.77	1.77	6.52	31.03
434.zeusmp	13.80	1.01	1.24	1.64	1.64	err	err
435.gromacs	1.37	1.11	1.25	1.52	1.60	5.64	20.44
436.cactusADM	2.47	1.02	1.44	3.00	4.66	6.15	err
437.leslie3d	11.50	1.02	1.13	1.48	1.48	4.83	12.00
444.namd	11.10	1.05	1.26	1.63	1.65	7.09	23.78
447.dealII	13.20	1.42	1.58	2.20	2.24	err	err
450.soplex	0.03	2.57	2.79	3.03	3.19	err	err
453.povray	0.50	1.62	2.11	2.68	2.88	11.15	52.52
454.calculix	0.05	2.38	2.75	3.24	3.22	16.57	44.01
459.GemsFDTD	2.10	1.32	1.45	2.00	2.07	5.62	17.14
465.tonto	0.80	1.43	1.71	2.10	2.25	8.39	31.54
470.lbm	3.27	1.00	1.02	1.10	1.11	3.24	11.77
482.sphinx3	1.45	1.30	1.49	1.84	2.00	8.90	34.69
Average	4.49	1.22	1.45	1.98	2.12	5.24	24.30
Geo. mean	1.68	1.43	1.67	2.21	2.36	7.13	26.39

Table 2: Performance evaluation using the SPEC CPU2006 benchmarks (*test* dataset). NAT shows native execution in seconds, the next four columns compare different memTrace configurations to NAT. The last two columns compare Valgrind nullgrind (VAL) and memcheck (VMEM) to NAT.

The WP configuration extends the MT configuration with the memlet for unlimited watchpoints. No watchpoints are set in this configuration, but the difference in execution time between configurations with set watchpoints and configurations without set watchpoints is negligible if no watchpoints are taken. If watchpoints are taken then the execution time of the watchpoint handlers must be added to the overhead as well. We measure the highest performance impact for the cactusADM benchmark with 3.11x performance impact compared to native execution due to the high frequency of memory accesses. The average overhead is 2.06x and the geometric mean is 1.97x. These two values show that the additional overhead for the user-defined memlet is low compared to the execution overhead of the baseline memory tracing framework.

The performance analysis shows that the overall overhead for the prototype of the lightweight memory tracing framework using cross-ISA translation (the MT configuration) is below 2.0x and that the additional performance impact for the execution of user-definable memlets is low (the WP configuration). Lightweight memory tracing is a technique that can be used in practice to trace every single memory access of an application using user-definable memlets with tolerable execution overhead.

6.2 Comparison to other systems

This section evaluates the prototype implementation of the memTrace technique with other, similar software products that are capable of memory tracing. We tried running the minemu 0.8 open-source version on a 64-bit Ubuntu 12.04 system. Unfortunately the current version of minemu crashes during the initialization of thread local storage of the SPEC CPU2006 benchmarks when running 32-bit x86 binaries on an x64 system.

6.2.1 Valgrind

We evaluate valgrind [18] version 3.7.0-0ubuntu3 as the second system in two configurations: nullgrind (VAL) to evaluate the Valgrind overhead and (VMEM) to evaluate Valgrind's memcheck overhead. Table 2 shows the timings of the SPEC CPU2006 benchmarks and compare different configurations against the native execution of the *test* dataset. The test dataset uses shorter input files and simpler problems. This comparison uses only the *test* dataset due to the higher translation overhead of Valgrind. The 434.zeusmp, 447.dealII, and 450.soplex benchmarks did not complete under Valgrind's nullgrind configuration and the 400.perlbench, 403.gcc, 434.zeusmp, 436.cactusADM, 447.dealII, and 450.soplex benchmarks did not complete under Valgrind's memcheck configuration. MemTrace uses the same configurations as in Section 6.1.

The evaluation for memTrace shows a similar picture like the performance analysis of the *ref* dataset. In general the overhead increases due to the fact that translated code in the code cache is reused less often. The geometric mean for the MT configuration is 2.21x and the average overhead is 1.98x (compared to a geometric mean of 1.86x and an average of 1.90 for the *ref* dataset).

Valgrind on the other hand exhibits an average overhead of 5.24x and a geometric mean of 7.13x for the *test* dataset in the nullgrind configuration. The nullgrind configuration is comparable to the ID configuration of memTrace and does not execute any memlets or other user-defined code. The memcheck configuration of Valgrind results in an average overhead of 24.3x and a geometric mean of 26.4x. The memcheck configuration is comparable to memTrace's WP configuration.

	1 WP [s]	10 WP [s]	100 WP [s]
GDB SW WP	180	330	1670
memTrace	0.01	0.01	0.01

Table 3: Evaluation of the microbenchmark in with the first watchpoint at the 1,000 element.

6.2.2 GDB

We use a CPU-bound microbenchmark to evaluate the performance of the watchpoint memlet compared to GDB. The microbenchmark sets W consecutive watchpoints in a large array and processes the array in multiple passes, where the n^{th} pass accesses the first n elements of the array. In each pass, the elements are accessed using several patterns: a forward linear sweep, a convolution, and a sparse backward sweep. The microbenchmark measures the time until the first watchpoint is hit and handled by the debugger.

To compare memTrace performance with hardware watchpoint performance we configure the microbenchmark with the first watchpoint at the $500,000$ array element (i.e., memTrace needs to execute a large amount of memlets that do not trigger a watchpoint). With one active watchpoint the hardware watchpoint configuration executes in 52.8 seconds while the memTrace implementation uses 80.5 seconds, resulting in 52% overhead compared to the hardware implementation. While hardware watchpoints support only up to 4 simultaneous watchpoints memTrace supports unlimited watchpoints at a constant overhead (10^4 watchpoints in 80.5 seconds and 10^8 watchpoints in 81.5 seconds). Even at 10^8 watchpoints the performance of memTrace remains stable.

Table 3 compares memTrace performance with the performance of GDB software watchpoints with the first watchpoint at the $1,000$ array element. Even for 1 GDB software watchpoint memTrace is 18,000x faster than software watchpoints. For 100 GDB software watchpoints memTrace is 167,000x faster. The prototype implementation of the memTrace watchpoint memlet fully supports the remote serial protocol of GDB and works as a fast drop-in replacement for the GDB software watchpoints.

6.3 Memory overhead

Table 4 presents an analysis of the memory overhead for the SPEC CPU2006 benchmarks when run natively and under the control of the memTrace prototype implementation. The table shows the peak amount of mapped memory of the benchmark. This benchmark measures the number of mapped memory pages, not the number of allocated memory pages. The allocated memory pages are a subset of the mapped memory pages.

The memory overhead for binary translation only is low with an average of 9.2 MB and a maximum of 12.7 MB. Binary translation only needs few data structures (8 MB for the mapping table plus data structures for the code cache, signal handlers, and trampolines). These numbers show that

Benchmark	NAT [MB]	ID [MB]	Ovhd.	WP [MB]	Ovhd.
400.perlbench	565.24	574.92	1.7%	1142.16	102.1%
401.bzip2	628.27	636.76	1.4%	1265.34	101.4%
403.gcc	84.05	96.74	15.1%	186.92	122.4%
429.mcf	856.22	864.64	1.0%	1721.12	101.0%
445.gobmk	38.66	48.66	25.9%	89.76	132.2%
456.hmmer	21.07	29.61	40.5%	51.11	142.6%
458.sjeng	192.09	200.64	4.4%	393.17	104.7%
462.libquantum	114.02	122.44	7.4%	236.78	107.7%
464.h264ref	81.00	89.97	11.1%	172.15	112.5%
471.omnetpp	119.57	129.02	7.9%	250.28	109.3%
473.astar	140.52	149.00	6.0%	289.96	106.3%
483.xalancbmk	329.77	340.42	3.2%	673.82	104.3%
410.bwaves	891.57	900.11	1.0%	1792.19	101.0%
416.gamess	655.31	665.31	1.5%	1323.43	102.0%
433.milc	687.87	696.41	1.2%	1384.78	101.3%
434.zeusmp	1136.98	1146.00	0.8%	2284.24	100.9%
435.gromacs	34.60	43.39	25.4%	78.87	127.9%
436.cactusADM	1017.70	1026.80	0.9%	2045.81	101.0%
437.leslie3d	141.02	149.68	6.1%	291.39	106.6%
444.namd	63.85	72.52	13.6%	137.12	114.7%
447.dealII	501.38	510.89	1.9%	1014.33	102.3%
450.soplex	509.26	518.04	1.7%	1028.18	101.9%
453.povray	21.84	31.10	42.4%	54.44	149.3%
454.calculix	179.23	188.73	5.3%	369.89	106.4%
459.GemsFDTD	845.69	855.01	1.1%	1702.32	101.3%
465.tonto	53.91	64.46	19.6%	122.37	127.0%
470.lbm	426.93	435.35	2.0%	862.53	102.0%
482.sphinx3	57.81	66.59	15.2%	125.16	116.5%
Average	371.27	380.47	2.5%	753.20	102.9%
Geo. mean	201.55	219.56	8.9%	413.58	105.2%

Table 4: Memory consumption in megabytes of the SPEC CPU2006 benchmarks (NAT) and additional memory consumption of different configurations of the memTrace prototype implementation. ID represents a binary translation only configuration.

the memory overhead for cross-ISA binary translation is low.

The last column of Table 4 shows the memory overhead of the WP configuration. The amount of mapped memory is roughly doubled due to the shadow memory region.

7 Related work

There are several areas of related work that are relevant for lightweight memory tracing. Binary translation is needed to dynamically weave the memlets into the executed application code. The following sections discuss different systems for binary translation and different systems that implement some forms of memory tracing.

7.1 Binary translation

Binary translation enables late code modification to, e.g., instrument a binary application, to offer late code optimization, or to execute an application on a different ISA than it was originally compiled for.

Full-ISA emulation is too slow for real-world scenarios and mostly used for evaluation of new hardware features. Efficient binary translation is implemented using either table-based approaches or IR-based approaches.

Same-ISA binary translation translates an application to the same ISA (x86 to x86). A drawback of same-ISA translation is the register pressure on x86. Only 8 general purpose registers are available for x86 applications and only 6 or 7 registers are available for general computation (depending on the calling conventions). Memlets used for memory tracing need to execute additional computation for each memory access, starving the register allocator even further.

IR-based binary translators translate the application by using a traditional compiler approach. The binary translator transforms code into an IR, adds the desired instrumentation, and generates machine code for the desired platform. Translation is either dynamic like in a just-in-time compiler or static ahead-of-time. DynamoRIO [5], PIN [15], QEMU [3], and Valgrind [18] are dynamic IR-based binary translators. The IR-based approach enables compiler optimization to produce high-quality code at some translation cost.

Dynamic table-based binary translators (e.g., HDTrans [25], fastBT/libdetox [20, 21], or StarDBT [28]) use translation tables to decode original instructions and to generate translated instructions. The advantage is the low-overhead translation speed combined with reasonable code quality.

StarDBT [28] and QEMU [3] are two binary translation systems that support cross-ISA translation. StarDBT translates x86 code to x64 code and QEMU translates (almost) any ISA to (almost) any other ISA.

MemTrace is a cross-ISA table-based dynamic binary translator that translates user-space applications from x86 to x64. The binary translator component offers near-native performance. The StarDBT binary translator is similar to our binary translator but uses two compilation stages (baseline and optimized) while memTrace uses only one fast table-based translation scheme. In addition, memTrace allows the definition of user-defined memlets that may use fixed registers to speed up memlet execution.

7.2 Memory tracing and watchpoints

Memory tracing allows the execution of memlets for each memory access. A baseline memory tracing infrastructure is needed to implement higher-level memlets like watchpoints, or taint checking.

Greathouse et al. [12] present a case for unlimited watchpoints and light-weight, hardware-assisted memory tracing. They reason that additional hardware is needed to achieve low overhead for unlimited watchpoints. MemTrace shows that cross-ISA translation realizes low-overhead memory tracing (and watchpoints) for x86 applications when executed on modern processors that support x64 extensions.

Metric [16] is a memory tracing framework that collects and stores selected memory access traces. Memcheck [17],

System	Arch.	Underlying BT	Shadow memory	Overhead
memTrace	x86 to x64	libdetox	1 byte per byte	2.0x for SPEC CPU2006
Libdft [14]	x86 to x86	PIN	flexible	1.14x to 10x slowdown SPEC CPU2000
Minemu [4]	x86 to x86	dynamic BT, no SSE[a]	1 byte per byte	2.4x for SPEC INT2006
PTT [9]	x86 to x86	QEMU	32-bit vector per byte	21.1x for compression benchmark
Saxena et al. [24]	x86 to x86	static BT	1 bit per byte	1.9x (stack only) to 2.8x (SPEC INT95 subset)
Panorama [30]	x86 to x86	QEMU	4 byte pointer per byte	20x on selected benchmarks
Dytan [7]	x86 to x86	static BT	1 bit vector per byte	30x to 50x for gzip
LIFT [23]	x86 to x64	StarDBT	1 bit per byte	1.7-7.9x, 3.6x for SPEC INT2000
Argos [22]	x86 to x86	QEMU	1 bit per byte (phys. mem)	"at least 16x overhead"
Xentaint [13]	x86 to x86	Xen and QEMU	1 bit per byte	61.5x to 88.4x for micro-benchmarks
Vigilante [8]	x86 to x86	static BT on start-up	1 bit per 4k page	no numbers on performance overhead reported
Taintcheck [19]	x86 to x86	Valgrind	4 byte pointer per byte	1.5x to 40x
Suh et al. [26]	Alpha	HW extension	1 bit per page/quad word/byte	1.44% for SPEC CPU2000

[a] Minemu internally uses the SSE registers and cannot support any SSE instructions in applications. Modern compilers use SSE instructions to speed up memory transfers, for vectorization, and for floating point computation.

Table 5: Comparison of different taint checking and dataflow analysis systems.

Umbra [31], EDDI [32], and Dr. Memory [6] are four frameworks for memory tracing that use same-ISA binary translation to add hard-coded memlets for watchpoints. Memcheck builds on Valgrind and reports an overhead of 22.2x for the SPEC CPU2000 benchmarks. Umbra, EDDI, and Dr. Memory build on DynamoRIO. Umbra reports an overhead of 2.33x for SPEC CPU2006 for memory tracing of an x64 application; an example tool that extends Umbra with a memlet that monitors thread's memory accesses imposes a 6.49x overhead for a set of benchmarks. EDDI reports an overhead of 2.59x for 0 watchpoints and 3.68x for watching the complete data region on the SPEC CPU2000 benchmarks in the FI configuration. The PI configuration of EDDI only reports on a subset of the SPEC CPU2000 benchmarks. Dr. Memory reports a slowdown of 10.2x for the SPEC CPU2006 benchmarks. Umbra implements memory tracing without additional memlets; memcheck, EDDI, and Dr. Memory add hard-coded instructions into the executed application code to check memory accesses for validity.

MemTrace improves on related work by offering user-definable memlets that implement high-level memory checkers and offers better performance than previous solutions: memTrace reports an average overhead of 2.06x and a geometric mean of 1.97x for tracing all memory accesses of all SPEC CPU2006 benchmarks.

7.3 Taint checking and dataflow analysis

Taint checking and data flow analysis extend memory tracing and analyse the flow of data inside an application. Every memory cell and every register has an associated tag. Taint checking uses a single taint bit per address while dataflow analysis supports multiple different tags. Compared to single-threaded approaches of other related work memTrace fully supports memlets for concurrent threads.

Some of the systems in the following list use taint checking or dataflow analysis as a technique in their system. Table 5 focuses on the taint checking or dataflow analysis component of the presented systems.

MemTrace does not change the address space layout of the original application, all data of the memlets is stored at a higher location in the 64-bit memory space. This design decision solves the problem of accesses to the shadow memory by the application. For the shadow memory itself memTrace uses 1 byte per byte, enabling threads to update the (shared) shadow memory data structure concurrently without locking. Only if memlets rely on bit-granularity then the programmer must add a locking scheme to ensure correctness.

8 Conclusion

This paper presents memTrace, a technique for dynamic lightweight memory tracing for unmodified binary applications. This technique adds shadow memory and state for each memory address of an application and allows the execution of user-defined memlets to inspect memory accesses.

The practical value of memTrace is demonstrated by the implementation of two memlets: a memory checking memlet that allows the debugging of memory errors and a memlet that allows an unlimited number of watchpoints in a running application. We evaluated the prototype implementation and show that the overhead for SPEC CPU2006 is low with a geometric mean of 1.97x and an average of 2.05x.

The open source release of the memTrace prototype is available at `http://nebelwelt.net/projects/memTrace` and can be used to implement other memlets, e.g., for taint checking, dataflow analysis, or control flow integrity checks.

Acknowledgments

We thank the anonymous reviewers for their comments, Albert Noll for his comments on an early draft of this paper, and Jonas Pfefferle and Tobias Hartmann for working on a same-ISA version of a simple memory tracing infrastructure.

References

[1] ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., AND STICHNOTH, J. M. Fast, effective code generation in a just-in-time java compiler. In *PLDI'98* (1998), pp. 280–290.

[2] ALPERN, B., BUTRICO, M. A., COCCHI, A., DOLBY, J., FINK, S. J., GROVE, D., AND NGO, T. Experiences porting the jikes rvm to linux/ia32. In *Java Virtual Machine Research and Technology Symposium* (2002), pp. 51–64.

[3] BELLARD, F. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC* (2005), pp. 41–41.

[4] BOSMAN, E., SLOWINSKA, A., AND BOS, H. Minemu: the world's fastest taint tracker. In *RAID'11: Proc. 14th conf. on Recent Advances in Intrusion Detection* (2011), pp. 1–20.

[5] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO '03* (2003), pp. 265–275.

[6] BRUENING, D., AND ZHAO, Q. Practical memory checking with dr. memory. In *CGO'11* (2011), pp. 213–223.

[7] CLAUSE, J. A., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Intl. Symp. on Software Testing and Analysis* (2007), pp. 196–206.

[8] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A. I. T., ZHOU, L., ZHANG, L., AND BARHAM, P. Vigilante: End-to-End Containment of Internet Worms. In *SOSP'05* (2005), vol. 39, pp. 133–147.

[9] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L. L., AND MCCAULEY, M. Towards practical taint tracking. Tech. Rep. UCB/EECS-2010-92, EECS Department, University of California, Berkeley, Jun 2010.

[10] GDB. GDB remote serial protocol. `http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html`, 2010.

[11] GLOGER, W. Dynamic memory allocator implementations in linux system libraries. `http://www.dent.med.uni-muenchen.de/~wmglo/malloc-slides.html`, May 1997.

[12] GREATHOUSE, J. L., XIN, H., LUO, Y., AND AUSTIN, T. A case for unlimited watchpoints. In *ASPLOS'12* (2012), pp. 159–172.

[13] HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. Practical taint-based protection using demand emulation. In *EuroSys'06* (2006), pp. 29–41.

[14] KEMERLIS, V. P., PORTOKALIDIS, G., JEE, K., AND KEROMYTIS, A. D. libdft: practical dynamic data flow tracking for commodity systems. In *VEE'12* (2012), pp. 121–132.

[15] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI'05* (2005), pp. 190–200.

[16] MARATHE, J., MUELLER, F., MOHAN, T., MCKEE, S. A., DE SUPINSKI, B. R., AND YOO, A. Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. *ACM Trans. Program. Lang. Syst. 29*, 2 (Apr. 2007).

[17] NETHERCOTE, N., AND SEWARD, J. How to shadow every byte of memory used by a program. In *VEE'07* (2007), pp. 65–74.

[18] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07* (2007), pp. 89–100.

[19] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS'05* (2005).

[20] PAYER, M., AND GROSS, T. R. Generating low-overhead dynamic binary translators. In *Proc. 3rd Annual Haifa Experimental Systems Conf.* (2010), SYSTOR '10, ACM, pp. 22:1–22:14.

[21] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments* (2011), pp. 157–168.

[22] PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys'06* (2006).

[23] QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO'06* (2006), pp. 135–148.

[24] SAXENA, P., SEKAR, R., AND PURANIK, V. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *CGO'08* (2008), pp. 74–83.

[25] SRIDHAR, S., SHAPIRO, J. S., AND BUNGALE, P. P. HDTrans: a low-overhead dynamic translator. *SIGARCH Comput. Archit. News 35*, 1 (2007), 135–140.

[26] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. Secure program execution via dynamic information flow tracking. In *ASPLOS'04* (2004), pp. 85–96.

[27] VMWARE. Software and hardware techniques for x86 virtualization. `http://www.vmware.com/files/pdf/software_hardware_tech_x86_virt.pdf`, 2009.

[28] WANG, C., HU, S., KIM, H.-S., NAIR, S., BRETERNITZ, M., YING, Z., AND WU, Y. Stardbt: An efficient multi-platform dynamic binary translation system. In *Advances in Computer Systems Architecture*, vol. 4697. 2007, pp. 4–15.

[29] WIMMER, C., AND FRANZ, M. Linear scan register allocation on ssa form. In *CGO'10* (2010), pp. 170–179.

[30] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS'07* (2007), pp. 116–127.

[31] ZHAO, Q., BRUENING, D., AND AMARASINGHE, S. Umbra: efficient and scalable memory shadowing. In *CGO'10* (2010), pp. 22–31.

[32] ZHAO, Q., RABBAH, R., AMARASINGHE, S., RUDOLPH, L., AND WONG, W. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *CC'08*.