Trace format

Binary file format description, rationale and its uses

First release version: October 2013

Note: although the binary trace format described in this document is a pretty stable version, it is bound to evolve. As of such, both this document and the underlying file format are subject to change.

Table of Contents

1 Introduction

This document describes trace files format and its design rationale. It also explains the underlying data format, its logic, and roughly how a trace analyzer is expected to use trace data. All this extra information is threaded around the file format itself.

The reader is assumed to possess sufficient knowledge on *Microsoft Windows* architecture and on *Intel* x86 and x64 architecture and assembly. For thorough information on the latter, see *Intel® 64 and IA-32 Architectures Software Developer Manuals*:
http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

1.1 About traces

A raw trace is first generated from a running application with a home-made *Pin tool* called *UMTracer*, before being indexed and converted to be more usable. Raw traces come in two parts:

- Instrumentation trace:
    - Akin to static code analysis
    - Records instructions
        - Addresses and code bytes, not their execution.
        - Recording occurs when those instructions are instrumented, i.e. “augmented” to generate the execution trace when executed
    - Special events (load/unload modules)
- Execution trace:
    - Akin to dynamic code analysis
    - Generated by instrumented code as it runs
    - Records instructions execution
        - Registers and referenced memory values
    - Special events
        - Kernel and user mode transitions (system calls, context changes, exceptions)
        - Thread creation and termination
    - Sequential number on each record

A side tool, UMTracerRemote allows turning instrumentation on and off during the process. This only affects *instructions* instrumentation since UMTracer always record system calls and other special events. For general information about code instrumentation, see *Pin* documentation and papers:
http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

UMTIndex, another home-made tool, converts a raw trace into an *indexed trace* (simply called *trace* in the remainder of this document). It validates the raw trace, recovers what it can if the trace is (abrupt process or OS termination for instance), and reorders the execution trace according to sequence numbers. The result is stored into a few binary files.

2 Format design rationale

The trace format design was driven by a few use cases that all share a similar base scenario:

- A raw trace is generated while running an application with *UMTracer*
- The raw trace is then verified and indexed
    - Can be done at a later time, on a different machine, in a batch, etc.
- The trace is expected to contain
    - Several billions of small (< 100 bytes) records
    - Millions of widely variable size (10 bytes to 100 MB) records.
- One or many trace analyses then are performed offline
    - The trace is read-only
        - Annotations and other generated data are stored elsewhere.
    - Analyses are typically of sequential nature
        - Records are processed in chronological or control flow wise order

Given the above and the sheer amount of data to process, data fetching cost can be relatively significant to trace analysis intrinsic complexity. Data fetching performance boils down to a trade-off between:

- Easy data access:
    - Fully expanded values (propagated leading zero or sign bit)
    - Pre-computed pointers, offsets, etc.
        - Information redundancy in various forms
    - Structured row-column tables.
- Fast data access:
    - Related data are physically close to each other
        - Disk buffers and memory cache friendly
        - Semi-structured stream-like records (variable fields and size)
    - Minimal file size:

- Little or no redundancy
- Data reuse on identical chunks of data across records (multiple file “pointers” to the same location)
- Disk access can be a bottleneck, unless running on powerful systems (high transfer speed to read trace into memory, and enough memory to fit the whole trace, perhaps 100 GB in size).

Therefore, smaller data collections (instructions, modules, thread descriptions, etc.) favor ease of access, while potentially larger collections (executed instructions) lean towards fast data access

The trace can be stored either in a typical DB or in home-made binary files. The latter was chosen for a few reasons:

- DB engines limitations:
  - Existing DB engines can be pretty slow when it comes to requesting billions of records in a near sequential fashion.
  - The mere overhead of generating/decoding DB requests and copying/marshalling data may be 10 to 100 times higher than data processing itself.
- Custom format processing efficiency:
  - Flat binary files with (known) custom structures can be mapped into the trace analyzer process memory, in whole or in part.
  - A trace analyzer implemented in a language that provides full control over memory layout of structured types (C or C++ for instance) allows direct mapping of implementation language structures over trace data without any copy or marshalling.

Moreover, currently envisaged trace analysis use cases do not perform typical DB operations besides requesting records by their index. Therefore, the lack of DB requests versatility does not warrant reinventing the wheel in trace analyzer implementations. Nevertheless, should the need arise for DB-oriented operations, a new front-end could be added to UMTIndex to generate a proper DB from a raw trace.

3 Data types

This section describes data type names and conventions used throughout this document.

Some general considerations:

- Constants
    - Unless otherwise noted, constants in this document are in decimal.
    - Hexadecimal constants are prepended with `0x`, such as `0x1234`.
- Signedness
    - Types can be explicitly specified as signed or unsigned.
    - When unspecified, sign does not matter and trace reader implementation is free to use whatever representation seems fit. Only the actual binary value (bits) matters.
    - Some constants in this document may be written as signed/unsigned for the sake of readability. For instance, a 32-bit value written as `-1` means `0xffffffff`. Unless otherwise specified, it does not matter if the trace reader implementation uses it as a signed (`-1`) or unsigned (`4294967295`) value.
- Byte ordering
    - Little-endian (Intel x86 and x64)
    - Low order byte is stored first, and high-order byte is stored last. Value `0x12345678` is stored as 4 bytes: `0x78 0x56 0x34 0x12`.
- Size vs count/length
    - Throughout this document, unless otherwise noted,
        - "size" is in bytes,
        - "count" or "length" is a number of elements in an array or a list,
    - `sizeof(`type`)` denotes the size of a type, most often a composite type (see Structures later in this section).

3.1 Basic types

- `i8`, `i16`, `i32`, `i64`: integral values (8 to 64 bits) that can safely be implemented with either signed or unsigned integral values. Such an unspecified signedness type is typically used in two cases:
    - The value is raw binary contents or some ID, and is not used in any arithmetic operation.
    - It represents an unsigned quantity, such as a string length, but the highest half of the value's range (with the highest bit set, which would otherwise be signed) will never occur. For example, a string length of the type `i64` means it is safe to assume that the value will remain within [`0`, $2^{63}-1$].
- `si8`, `si16`, `si32`, `si64`: SIGNED integral values (8 to 64 bits)
- `ui8`, `ui16`, `ui32`, `ui64`: UNSIGNED integral values (8 to 64 bits)
- `bool`: one-byte value that is either 0 or 1.
- `ac`: ASCII character, similar to `i8`.
- `uc`: UTF-16 Unicode "code unit" (not "character", as some characters are composed using more than one code unit).
    - Similar to `i16`.
    - See http://msdn.microsoft.com/en-us/library/windows/desktop/dd374081.aspx about code units.

3.2 File "pointers" (offsets)

- `Type *`: Signed 64-bit file offset into the same file. It is relative to a specific point in the file, given in that field's description. The "type" does not appear in the data itself, and complex types can be omitted in the description to improve readability.

3.3 Arrays

- `Type[count]`: Array of `count` elements of type `Type`.
    - All elements are packed without extra padding. When `Type` explicitly contains padding (see `pad` below), this padding is kept, even if it occurs at the end of `Type`.
    - Elements are indexed from `0` to `count-1` inclusively, and found at offset `i * sizeof(Type)` after element `0`.

    - Total size in bytes of the array is always `count * sizeof(Type)`.
    - If the element count is not specified in the type description, it must come from another data field.

- `pad[size]`: "size" bytes of padding. Those bytes contain garbage and are meant for alignment purpose only (may improve performance or ease structure definitions).
- `buf[size]`: "size" bytes buffer of undefined type. Within a table, this usually holds several variable-size data pointed to (via file pointers) by other table fields.

3.4 Structures

As with many programming languages, structures are tuples whose number of elements and their respective type are fixed. However, trace structure elements may also have a variable size when specified as such. In such cases, the position of following elements, relative to the beginning of the structure, is not statically computable. Also, there is no implicit alignment of elements, on 4 or 8-byte boundaries for instance. Any alignment is explicitly included using a "padding" element.

Structures are defined by their name and size, then by the list of their element. Each element has an offset, type, name and description:

- Offset is in bytes, noted in decimal
    - From the beginning of the structure
    - The offset is always "offset of previous element" + `sizeof(`previous element type`)`.
    - Padding (for alignment purposes) when needed is always explicitly included
- Name is only used as a reference in descriptions of other fields, for the sake of clarity. It does not appear anywhere within the data itself.
- Description may use the following abbreviations:
    - EID: Unique execution record ID (64 bits), as found in the execution trace.
        - Record IDs are unsigned numbers in theory, although they will never reach 2^{63} in supported use cases. So signed 64-bit values will do just fine too.
    - FP: File pointer. Relative to a specific point in the file, given in the description.

3.5 Files

File contents is described like any structure type.

4 Trace format overview

4.1 Components

`Execution` table is the main table. It contains all executed instructions of the trace, chronologically ordered (to some extent, see Error: Reference source not found Error: Reference source not found). Its records directly or indirectly refer to other tables. Content of those tables is summarized below, and Error: Reference source not found depicts basic relation between all tables.

- **`System calls`**
  - Extra contextual information recorded at system calls entry and exit points.
- **`Instructions`**
  - All instructions ever instrumented during trace recording
    - Basic information: module it belongs to, address, code bytes
    - Decoded instruction: explicit and implicit operands, constants, text disassembly
    - Data pattern for recorded values in execution trace
  - Generated from the raw instrumentation trace
- **`Modules`**
  - DLLs loaded at some point into the traced process
  - Information about them (file name, size), not their actual contents
- **`Threads`**
  - Basic information on all threads of the traced process
  - Does not include control-flow information, which is found in the execution table

Two tables are not referred to by others, but will tell a trace analyser how to process data:

- **`Context map`**
  - Register values are sometimes dumped all at once in an execution record as a monolithic "context"
  - This map tells where each register within that context is.
- **`Address space usage`**

- List of all memory address (4kb page ranges) ever referred to by executed instructions.

Another file, ***trace.xml***, also provides information on how to process data (which "enum tables" to use) as well as some meta-data (e.g. traced process command-line).

Finally, a few "enum" tables, external to the trace, may be needed to trace analyzer tools developers. They provide a somewhat meaningful representation (literal names) of numerical IDs used in other tables (system call IDs, instruction mnemonics, etc.).

More details on each table are presented in their respective section of this document.

4.2 Generic table file format

Table file format comes in two flavours:

- Self-indexed, fixed-size records
- Variable-size packed records list

4.2.1 Fixed-size records

This file format is mostly used by tables that are expected to be small (typically < 1GB) and mapped or loaded all at once in memory for processing. All tables but `Execution` and `System calls` use this format.

The table is merely an array of records, with direct access to any record by its 0-based ID. If variable-size fields are needed, they are located in an extra area at the end of the file and are pointed to from fixed-size fields in the table. Trace tables stored in that format share a common header and the following layout, where `record` is table-specific:

Offset	Type	Name	Description
`0`	`i32`	`HeaderVer`	Header version number. Layout of the remainder of this header depends on its version. Only version 0 is currently supported for all tables.
`4`	`i32`	`RecordVer`	Record layout version number. This refines the `record` type to use (which type or which version of this type). All known record types in this document only support version 0.
`8`	`i64`	`RowCount`	Number of records in the table
`16`	`i64`	`RowSize`	Size of each record, always `sizeof(record)` for a given `RecordVer`
`24`	`record *`	`TableFP`	File position (absolute, from start of file) of the first record (ID = 0) of the table.
`32+`	…	…	Table-specific additional header information.
`TableFP`	`record [RowCount]`	`Table`	Array of `RowCount` contiguous records.

4.2.2 Variable-size packed records list

This file format does not have a specific structure. Stream-like records (fields may depend on some header values) are simply collected back to back (ordered or not) in a file. Records are not accessed directly by their ID. A second file provides an offset index where each entry points (absolute file offset) to the corresponding record in the former file.

`Execution` and `System calls` tables use this format.

5 Common structure types

A few structure types are not intrinsically related to any table.

cStr: null-terminated character (8 bits) string with explicit size
Size: **Length** + 9

Offset	Type	Name	Description
0	i64	Length	String length in characters (bytes), not including trailing null
8	ac[Length+1]		String contents followed by a null character (0)

uStr: UTF-16 null-terminated string with explicit size
Size: **Length***2 + 10

Offset	Type	Name	Description
0	i64	Length	String length in UTF-16 “code units” (not characters), not including trailing null. The actual number of Unicode characters is always lesser or equal to **Length**. About “code units”, see http://msdn.microsoft.com/en-us/library/windows/desktop/dd374081.aspx
8	uc[Length+1]		String contents followed by a null character (16-bit 0)

6 Execution table

6.1 Underlying data format

`Execution` table contains every executed instruction in chronological order (or almost, see Error: Reference source not found Error: Reference source not found), i.e. their ID is an execution sequential number. It also contains references to some execution events such as thread creation and system calls. Records therefore have a "type" and some data fields that depend on that type. They also share some common fields.

6.1.1 What is in an execution record

Informally, records contain:

- Thread ID: refers to a thread record in the **`Thread`** table.
- Thread control-flow: IDs of the previous and next execution records according to the control-flow for the current thread
- Flags: bit flags providing generic information about the execution at that point (32/64-bit, instrumentation turned on/off, etc.)
- Type: instruction, system call entry/exit, etc.
- Type-dependant data.

For most record types, type-dependant data provides a complete "local context", that is, recorded values for all registers and/or memory locations relevant to that instruction. For example, a system call or context change records provide a full register context dump, and instruction-type execution records contain all operands values, both explicit and implicit. Therefore, in most cases, a trace analyzer does not have to maintain a virtual machine state (registers and memory contents) to analyze a record. In other words, as far as a virtual machine state is concerned, records are pretty much self-contained. This may imply some redundancy and larger table files, but may greatly improve analysis performance and simplicity afterwards.

Still, maintaining a virtual machine state or searching through previous and next records may be required for replay-debug scenarios or thorough system call analysis. In the latter case, in a user-mode trace, system calls are akin to mega-instructions with several operands and probably several side effects as well. Basically, operands are the system call parameters, and those are included in execution records. However, semantically, operands also refer to:

- The various OS-specific structures they might point to in memory (which in turn may contain pointers to other structures, and so on).
- Kernel objects, via handles.

Neither UMTracer nor UMTIndex tools deal with those for now, except for a very few experimental cases that are not described in this document (but for which execution records may contain a few extra values recorded from memory). This would require specific handling for each of the few thousands known system calls as well as the hundreds (or more) of data structures used to make those calls. Nevertheless, a trace analyzer can implement some workarounds:

- Data structures in memory (system call parameters): if their values are really used by the user-mode calling code, those values will be read or written in instructions preceding or following a call. The will appear in their own record as with any memory access.

- Kernel objects: those are opaque to user-mode code, although in several cases the Win32 native API provides ways to query them. Obviously, this cannot be done offline from a trace. Only the tracer could do it live. The best a trace analyzer could do is to track a handle origin (a call to `NtOpenFile` for instance) and try to figure out what it refers to. Nevertheless, tracking what happens to that object from that point has its limits: handles may refer to objects shared with other processes and internal object state may thus be modified without any hint in the recorded trace.

6.1.2 Organizing huge record collections

Records' size varies from one to the other, so they need a "file offset and size" index. Semantically, this looks like the following ("educated random" values):

ID	Offset	Size		Thread	Type	Flags	Prev	Next	Data
0	16	18	→	1	Begin	0x3	NULL	1	
1	34	24	→	1	Ins	0x3	0	3	
2	58	19	→	2	Begin	0x3	NULL	4	
3	77	220	→	1	Ins	0x1	1	6	...
4	297	23	→	2	Ins	0x1	2	5	
5	320	17	→	2	Ins	0x1	4	7	
...	...	...		...	...	...	...	...	...

In reality, such an index takes about 25% of the overall table size and Prev/Next together take another 25%, meaning only half of the table contains actual execution data. This would be worse should the ID be stored in the table as well. Since the execution table can be pretty large, a lot of data must be read even though only part of it might be used. The following optimizations were applied to alleviate this issue:

- Remove "Size"
    - Records are stored in order (offset is strictly increasing) and size of a record can be inferred with the offset of the following record.
    - "Offset" column contains an extra offset, pointing right after the last record
- Move "Prev" and "Next" columns into a separate file.

- Trace analyzers that only rely on sequential execution order have no use of thread control-flow and may thus entirely avoid reading it.

6.2 File format

Execution table is split into three files, as shown in Figure 1:

Figure 1 – Execution table files

exec.vtable: packed collection of variable-size execution records, less thread CF columns
Size: variable

Offset	Type	Name	Description
0	i32	Version	Version number (only version 0 is currently supported)
4	pad[4]		
8	i64	RecordCount	Valid IDs for all three files are in [0, RecordCount [
Offset[i]	execRec		Variable-length record (see execRec structure below) for each i in [0, RecordCount [. Records are sorted by their ID.

exec.prev_next.column: thread CF information, i.e. "Prev" and "Next" columns of the table.
Size: RecordCount * 16

Offset	Type	Name	Description
0	i64Pair[RecordCount]	PrevNext	Each pair PrevNext[i] holds respectively the IDs of the previous and next records, thread CF-wise, for record i in [0, RecordCount [. ID −1 is used when there is no previous or next record.

exec.offsets: Absolute file offsets into ***exec.vtable*** for each record.
Size: (RecordCount + 1) * 8

Offset	Type	Name	Description
0	i64[RecordCount+1]	Offset	Absolute file offsets into ***exec.vtable*** for each record, and an extra file offset pointing after the end of the last record. Any record i is always fully contained in ***exec.vtable*** within [Offset[i], Offset[i+1] [.

6.3 Execution record format

execRec: execution record, combining a common header and variable type-dependent data
Size: variable (at least 6)

Offset	Type	Name	Description
0	i32	ThreadID	Thread ID (in Thread table) this record belongs to.
4	i8	Type	Record type, see type descriptions below
5			Combination of bit flags, see below

	i8	Flags	
6	?	Data	Type-dependent data, see type descriptions below

execRec.Flags can be a combination of the following values:

- **0x01**: 64-bits. Instruction or special event occurred within the context of a 64-bit mode (aka long-mode) code segment. 32-bit otherwise.
- **0x02**: Instruction instrumentation was OFF before execution of this record (see Introduction Introduction). When set, context values (register values and perhaps memory references) from previous records cannot be assumed to be correct for the current record. However, context values in current record data are valid for that record, and can be propagated to the next record if it does not have this flag set.
- **0x04**: non-executed conditional instruction. Applies to the following conditional instructions: **cmov**$_{cc}$, **fcmov**$_{cc}$, **j**$_{cc}$ (conditional jumps) and **rep**$_{cc}$-prefixed instructions. This flag is set when the condition **cc** was false and the remaining of the instruction was not carried on. All input and (unchanged) output operand values nevertheless appear in the current record data.
- **0x08**: set on the first iteration of a **rep**$_{cc}$-prefixed instruction.
- **0x10** to **0x80**: set when a low-level exception occurred during instruction execution, such as a memory access violation. Flags do not reveal the kind of exception but tell when it happened. Also, since the instruction and/or associated trace recording code did not complete, input and/or output values were not recorded either (although space was reserved for them in the record data, which then contains garbage).
    - **0x10**: occurred while recording input operands. Both input and output operands values contain garbage.
    - **0x20**: occurred during instruction execution. Input values are properly recorded, but output values are not.
    - **0x40**: occurred when tracing output memory operands. For some reason, although the instruction executed fine, output operands could not be read back to be recorded.
    - **0x80**: internal error, probably a bug in the tracer tool.

Fields in **execRec.Data** depend on **execRec.Type**. The following table list all possible types, their meaning and the fields found in **execRec.Data**. Some fields are structures and those structure types are defined after the table. Multiple fields, if there are, are always packed without any padding.

Type	Description		
	Field type	**Field name**	**Field description**
0	**Instruction**		
	i64	InsID	Corresponding instruction record ID in Instruction table
	buf[?]	OperVals	Values of all instruction operands, both implicit and explicit. Provides the instruction "local context" (see What is in an execution record What is in an execution record). The size and data layout of the buffer is given in the instruction record.
1	**Thread begin** Exactly once for each thread, before any other record for that thread.		
	rmContext	Context	Registers context at the thread start point, before the first user-mode instruction, as well as the thread environment block (TEB).
2	**Thread end** Exactly once for each thread that terminated before the traced application exited. This is the last record for a thread.		
	i32	ExitCode	Windows thread exit code
3	**Application end** The very last record of the trace. Present only if the traced application exited gracefully.		
	i32	ExitCode	Windows process exit code
4	**System call entry** Thread is about to enter kernel mode through one of the various kind of system calls. Typically follows a syscall/sysenter/int 2e instruction record.		
	i64	SyscallID	Corresponding system call record ID in System calls table.
	rmContext	Context	Value of all registers just before the call. Additional information, such as call parameters on the stack, is in the System calls table.
5	**System call exit** Thread just exit from kernel mode system call. Does not always immediately follow a system call entry record, and may not even match an entry at all (matching is done according to the stack pointer). Actually, system calls can be nested, involve user-mode callbacks, and can end on an exception or a context change. Also, some system calls, such as NtContinue, can shortcut a few system call exits at once.		
	i64	SyscallID	Corresponding system call record ID in System calls table, or –1 if there is no corresponding system call entry record (thus nothing in System calls table either). The latter case is semantically equivalent to a context change.
	rmContext	Context	Value of all registers right after the call. Additional information may be found in the System calls table.
6	**Skipped system call exit** A previous system call entry occurred, but starting from this record, the stack was unwound further than it would be on a normal system call exit (a "return" to the user-mode caller). In other words, the thread is not anymore in the context of that system call, and that call will never have a "System call exit" record.		
	i64	SyscallID	Corresponding system call record ID in System calls table.
8 9 10 11	**Context change – Asynchronous Procedure Call** **Context change – Exception Handling** **Context change – Callback** **Context change – Unknown reason** For one of the above reasons, the thread execution context just changed.		
	rmContext	Context	Value of all registers right after the context change, before carrying execution of the next instruction.

rmContext: full register context (all register values) and optionnaly some relevant values from memory.
Size: variable (at least 16)

Offset	Type	Name	Description
0	i32	MemCount	Number of elements in MemRefs
4	pad[4]		
8	buf[?] *	RegsFP	Points to a buffer that holds all registers value. Size and data layout of this buffer is given by the Context map (see section Context map). FP is relative to the start of this structure (its position in the file)
16	contextMem [MemCount]	MemRefs	Array of optional memory references added to the context.

contextMem: one memory reference, part of a **rmContext**.
Size: 24

Offset	Type	Name	Description
0	i64	Address	Memory address (high 32 bits are always 0 on 32-bit traces)
8	i64	Size	Size of the memory buffer
16	buf[Size] *	DataFP	Points to memory content. FP is relative to the start of the parent rmContext structure (its position in the file)

Note: Data pointed to from those structures is stored right after **rmContext** instance. That way, each record remains a monolithic stream of bytes. Any record **i** is always fully contained in ***exec.vtable*** within [**Offset[i]**, **Offset[i+1]** [, where **Offset** is found in ***exec.offsets***.

6.4 Execution order issues

The actual execution order that took place while tracing the application might slightly differ from recorded order. Instrumenting an instruction in the original application involves “recompiling” together that instruction and additional tracing code. A single instruction then turns into several instructions and in some cases, the original instruction itself is broken apart into a few instructions. A thread may therefore be pre-empted anywhere “during” an instrumented instruction.

Also, the tracer records values from memory before or after the actual instruction reads/writes that memory. Should that memory value be changed by another thread in between, there may be a discrepancy between the recorded value and the value used by the application. This is a rare event that is likely to occur only when tracing thread synchronization code (getting a lock for instance) or if there is some thread synchronization bug in the traced application. While this could be avoided in the tracing tool (*Pin* provides required functionalities), no efforts have been put so far regarding that issue. Should we ever need such accuracy (to track multi-thread bugs in applications for instance) we might improve the tracer accordingly.

Neither of those issues affects the binary file format. They may only produce wrong memory references values in very few execution record data.

7 Instruction table

7.1 Underlying data format

`Instruction` table is referred to by most execution records and is needed to do pretty much anything useful with those execution records. Instruction records are organized as an array of fixed-size records and two collections of variable-size sub-records pointed to by fields from the fixed-size part:

- Fixed-size part: static instruction information
    - Actual instruction: module it is from, address, code bytes, pointers to corresponding data in the two other sub-tables.
    - Recorded by the tracer as instructions are instrumented:
        - Not from some module-wide static code analysis.
        - Every instruction that is potentially about to be executed is instrumented, including self-modifying code.
        - Only instrumented instructions are recorded.
    - Instructions can be instrumented more than once for various reasons (namely the user turning instrumentation on an off during recording) and therefore recorded more than once into the table, on two distinct unrelated entries.
- Variable-size part: decoded instruction
    - Complete description of each instruction
        - Explicit and implicit operands (name and size)
        - Immediate values
        - Semantics revealed through enum values (instruction category, mnemonic and specific form), imported from Intel *XED* (see section Intel XED)
    - Data layout of operand values in execution record (the local context, see What is in an execution record What is in an execution record)
    - The same variable-size sub-record can be pointed to by several identical instructions from their respective fixed-size sub-record.
- Variable-size part: disassembly
    - Textual disassembly of each instruction, in typical Intel format.

- Identical instructions may point to the same disassembly string.

7.2 Decoded instruction

Intel instruction set is large, versatile and inherently complex. So is any structure that aims to uniformly describe all possible instructions. Figure 2 illustrates the basic relations between the above three sub-tables and execution records, with an emphasis on the "decoded instruction" sub-table. For the sake of clarity, some record fields were omitted or informally summarized.

Figure 2 – Simplified decoded instruction example

7.2.1 Local context slots

Local context slots are central to a decoded instruction. They list exactly all registers and memory references that are potentially used and/or modified. This also corresponds to the execution record "local context" buffer data layout. Slots are stored in a contiguous variable-size array and are grouped according to whether they refer to values before or after instruction execution. Pre-execution values of write-only operands are provided as a convenience to a trace analysis tool, although they are not actual inputs for that instruction. Slots referring to those values form a third slot group, between the two others. Besides this grouping, there is no particular order among slots.

Overlapping or "contiguous" operands can be merged in the execution record and sometimes in the slots as well (yet duplicates are allowed). For instance, if an instruction reads both `al` and `eax`, only `eax` value will be stored in the execution record and most often, only `eax` will appear in the slot list as well. Therefore, local context slots do not provide a clear picture of explicit instruction operands. This is instead provided by another level of indirection: explicit operands and pre-defined sub-operands (see the corresponding tables in Figure 2).

Explicit operands are just that: those that appear textually in disassembly or Intel documentation, in the same order. The current instruction set supports up to four explicit operands. Memory operands, whether explicit (`mov [rax], rdx`) or implicit (such as `[rdi]` in `stos` for instance), also have sub-operands in their "memory phrase" `seg:[base + scale*index + offset]`. Only one full memory phrase is supported per instruction, and the second memory operand, when there is, only has a base register. Each operand and sub-operand in use refers to one of the local context slots (including the two "constant" slots).

An explicit operand may refer only to a portion of the value the corresponding slot refers to. For instance, in `mov byte ptr [eax],al`, a single slot will be allocated for `eax` with a value size of `4`, while the second explicit operand will point to that slot but will specify a size of `1`. Explicit operands may also specify an additional offset into the value referred to by the slot. This is only required when an operand is `ah`, `bh`, `ch` or `dh` register and the corresponding slot refers to a larger enclosing register.

A trace analysis tool operating on instruction semantics would query instructions via its explicit operands and memory phrase sub-operands. But if only the list of input and output operands (or their values) is required, the tool may go directly for the local context slots.

7.2.2 Constant operands

Only explicit constants are found in the instruction record. Implicit constants (such as stack pointer offset on **push** or **pop**) are part of the instruction semantics and/or can be inferred from some fields of the instruction record (effective address or operand width, for instance). Constants do not need to be stored in the execution record, and are stored instead in two reserved operand slots.

If one of the explicit operands is a memory reference, the first constant slot is reserved for the memory phrase offset. Explicit constant operands (immediate values and branch offsets) simply point to either slot. In all cases, IP-relative offsets (in relative branches or 64-bit RIP-relative addressing) are converted to absolute constants using the address of the current instruction.

7.2.3 Memory operands

Operand name is a simple enum value. Most names are for registers (see section Intel XED) but a few refer to memory operands: **MEM0**, **MEM1**, **STACKPUSH**, **STACKPOP**. For these operands, the memory address is stored in the execution record along with the value. This alleviates the need to compute it from its constituents (memory phrase, stack pointer, etc.) although their values are all stored in the execution record as well. Note that memory-phrase/stack registers always have their "pre-execution" value (an address) stored in the execution record. Also, addresses in **FS** or **GS** segments are converted to flat linear addresses. Base address of each segment is provided in the local context as a convenience, but is not need to get the memory-based operand actual address.

Another special operand name, **AGEN**, is used solely for the **lea** instruction. It uses the same memory phrase sub-operands than **MEM0**, but does not have an associated memory value. Only the "address" is stored in the execution record, just like register values.

7.3 Instruction table file format

The **Instruction** table is stored in ***ins.itable***. Its header follows the fixed-size table format described in section Fixed-size records, where **record** type is **insRec**, described below. It also has the following table-specific fields in its header:

Offset	Type	Name	Description
32	buf[ExtraSize] *	ExtraFP	Absolute file pointer to the start of the two collections of all variable-size sub-records (decoded instruction and disassembly).
40	i64	ExtraSize	Total size of those two collections

Data at **ExtraFP** is not readily usable. It is rather accessed via instruction records, found at **Table[i]**.

In the following structures, a “local context slot index” (LC slot index) can be either an index into `decodedIns.LCSlots` array or one of the following special values:

- `SLOT_IMM0` (`0xfd`): value is found in `decodedIns.ConstSlot[0]`
- `SLOT_IMM1` (`0xfe`): value is found in `decodedIns.ConstSlot[1]`
- `SLOT_NA` (`0xff`): no corresponding slot, i.e. does not exist in local context

Also, operand names can either be physical register names (enum constants from Intel *XED*, **subject to change over time**, see section Intel XED) and/or one of the following special values:

- `MEM0` (`0xffff`): first memory-based operand of an instruction, referenced by a memory phrase of the form seg:[base + scale*index + offset], where most components are optional.
- `MEM1` (`0xfffe`): second memory-based operand of an instruction
- `AGEN` (`0xfffd`): address generator: integral result of a memory phrase, without reference to memory at that “address”.
- `STACKPUSH` (`0x007D`, from *XED*): output operand is on the stack at address `rsp`/`esp`/`sp - n`, where `n` depends on the operand size and pre-execution stack pointer value is used. Local context value in execution record provides the computed address like it does for any memory operands.
- `STACKPOP` (`0x007E`, from *XED*): input operand is on the stack at address `rsp`/`esp`/`sp`. Local context value in execution record provides this address like it does for any memory operand.
- `GS_BASE` (`0x0086`, from *XED*) and `FS_BASE` (`0x0087`, from *XED*): in a user-mode trace, it would be pretty useless to provide `fs`/`gs` selector values when those are used as segment overrides in memory phrases. Instead, their base linear address is provided.
- `INVALID` (`0x0000`, from *XED*): no associated register (to name a constant or non-existent operand for instance).

`insRec`: fixed-size part of an instruction record
Size: 64

Offset	Type	Name	Description
`0`	`i64`	`ID`	Self-ID in the current table.

8	i8	Type	Record type, provides a hint to the analyzer:
			• 0: normal instruction
			• 1: software interrupt (except alternate system call)
			• 2: 32-bit fast system call (syscall/sysenter)
			• 3: 64-bit fast system call (syscall/sysenter)
			• 4: alternate system call (int 0x2e on Windows)
			• 5: WOW64 system call
			• 6: other unknown system call
			• 8: NOP
			  ○ There are several ways with various mnemonics to encode a "no operation". All of them have this flag set.
			• 9: Prefetch
			  ○ Prefetched values are not recorded by the tracer and corresponding execution record operand values are likely to contain garbage.

9	bool	CSis64Bits	1 if the instruction was executed from a "long mode" 64-bit code segment, 0 otherwise.
10	i8	CodeSize	Number of valid bytes in CodeBytes array.
11	pad[1]		
12	i4	ModuleID	Module ID (in Module table) this instruction is from, or –1 if this instruction was from floating code in memory.
16	i8	Address	Address where this instruction was found (zero-extended on 32-bit applications). Note that addresses are not unique: • Instructions can be instrumented more than once. • Different code can be loaded/unloaded at a given address.
24	buf[15]	CodeBytes	Actual bytes of the instruction. Only the first CodeSize bytes are valid.
39	pad[1]		
40	cStr *	DisasmFP	Disassembly string of instruction, Intel style. FP is relative to ExtraFP in the file header.
48	decodedIns *	DecodedFP	Fully decoded instruction. FP is relative to ExtraFP in the file header. See decodedIns below.
56	pad[8]		

decodedIns: main (and somewhat monstruous) variable-size part of an instruction record

Size: variable (at least 88)

Offset	Type	Name	Description
0	i16	Category	Instruction category, from xed_category enum table. See section Intel XED.
2	i16	Mnemonic	Instruction mnemonic, from xed_iclass enum table. See section Intel XED.
4	i16	Form	Specific instruction form for the given mnemonic, from xed_iform enum table. See section Intel XED.

6	i8	Flags	Bitwise combination of zero or more of the following: • 0x01: has rep or repe/repz prefix • 0x02: has repne/repnz prefix • 0x04: has lock prefix • 0x08: instruction executes in 64-bit long mode, but operates on 32-bit register operands, in which case all output operands have their highest 32 bits zeroed • 0x10: instruction potentially reads from memory • 0x20: instruction potentially writes to memory
7	i8	ExpCount	Explicit operand count (0 to 4 incl.)
8	expOper[4]	ExpOpers	Array of explicit operands. Only the first ExpCount elements are valid. See expOper below.
56	i8	EffOperSize	Effective operand width in bytes (2, 4 or 8). Not the actual operands width, but rather related to instruction encoding and some zero/sign-extend rules. See Intel manual about “operand-size and address-size attributes”.
57	i8	EffAddrSize	Effective address size in bytes (2, 4 or 8). Not the actual size of code addresses, but rather related to instruction encoding and some zero/sign-extend rules. See Intel manual about “operand-size and address-size attributes”.
58	i16	LCSize	Size of local context data in execution record.
60	i8	FirstOutSlot	Index of the first LC slot for “before execution” value of write-only operand (see Figure 2), thus also the number of “before execution” values slots for read-only and read-write operands.
61	i8	FirstAfterSlot	Index of the first LC slot for after-execution operands (see Figure 2), i.e. after before-execution write-only operand slots.
62	i8	LCCount	Total count of LC slots (see Figure 2). Length of LCSlots array.
63	i8	Mem0SegSlot	LC slot index of the memory phrase segment override for any operand named MEM0 or AGEN.
64	i8	Mem0BaseSlot	LC slot index of the memory phrase base register for any operand named MEM0 or AGEN.
65	i8	Mem0Scale	Scale factor (1, 2, 4 or 8) of the memory phrase base register for any operand named MEM0 or AGEN, or 0 if there is not an index register as well.
66	i8	Mem0IndexSlot	LC slot index of the memory phrase index register for any operand named MEM0 or AGEN.
65	i8	Mem1BaseSlot	LC slot index of the memory phrase base register for any operand named MEM1.
68	pad[4]		
72	i64[2]	ConstSlot	Two constant slot values. Those are not lcSlot structures but only the actual value of an operand or memory phrase offset. If a MEM0 or AGEN operand is present in this instruction, ConstSlot[0] contains the memory phrase offset, and the number of significant bytes (up to 8) is given by lcSlot.AddrWidth of the corresponding operand slot. For explicit operands that directly refer to a constant slot, the number of valid bytes is given by the operand size.
88	lcSlot [LCCount]	LCSlots	Array of all local context slots, see lcSlot below.

`expOper`: description of an instruction explicit operand
Size: 12

Offset	Type	Name	Description
0	i16	RegName	Register name (from Intel *XED*, see section Intel XED) or one of the special values (see top of this section on page 22). May be different than `RegName` of the corresponding before/after slots since the latter may refer to a larger enclosing register. For instance, in `mov al, [ebx+eax]`, the first explicit operand may refer to an input slot that contains `eax` (no need to have `al` as a separate input slot).
2	i16	Size	Operand value size in bytes. Lesser or equal to `ValueSize` of the corresponding before/after slots. Even for memory operands, this is the size of the value in memory, not counting the address itself stored in the execution record. When `RegName` is `AGEN`, this is the size of the computed address. Note: while `Size` is most often a register size (up to 32 bytes for YMM registers), some instructions may have memory operands of about 600 bytes (`fxrstor` and relatives for instance).
4	i8	Flags	Combination of at least one of • `0x01`: operand is read • `0x02`: operand is written
5	i8	BeforeSlot	LC slot index for this operand "before execution" value. Never `SLOT_NA`.
6	i8	AfterSlot	LC slot index for this operand "after execution" value. `SLOT_NA` for read-only operands.
7	i8	BeforeOffset	Byte offset to add to `LCSlots[BeforeSlot].Offset` to get the actual offset, in the execution record local context, where the operand "before execution" value is stored. For register operands, this is always `0`, except for `ah`, `bh`, `ch` `dh` registers and only if the corresponding slot refers to a larger enclosing register (for instance, operand is `ah` but slot refers to `ax`, hence an additional offset of `1`). For memory operands, this is always `8` since a 64-bit address is stored before the actual memory value, even on 32-bits traces.
8	i8	AfterOffset	Same as `BeforeOffset`, for "after execution" operand value.
9	pad[3]		

`lcSlot`: local context slot, defines a portion of the local context data layout in an execution record
Size: 12

Offset	Type	Name	Description
0	i16	RegName	Register name (from Intel *XED*, see section Intel XED) or one of the special values (see top of this section on page 22).
2	i16	LargestEnclosing	Largest register enclosing `RegName`, according to the *absolute* enclosing register definition in section Context map. This comes handy for a trace analysis tool looking for a specific register (`al` for instance) in the LC. It can search by largest enclosing register (`rax`) instead of looking for all possible enclosing registers (`al`, `ax`, `eax`, `rax`)

4	i16	ValueSize	Operand value size. For memory operands, this is the size of the value in memory, not counting the address itself stored in the execution record. When RegName is AGEN, this is the size of the computed address. Note: while Size is most often a register size (up to 32 bytes for YMM registers), some instructions may have memory operands of about 600 bytes (fxrstor and relatives for instance).
6	i16	Offset	Offset, relative to the start of the execution record LC data, where the register/memory value (ValueSize bytes wide) is found. For memory operands, a 64-bit address (zero-extended from original if needed) precedes the value.
8	i8	AddrWidth	0 for non-memory operands. Otherwise, the significant size of the address value in the execution record LC. Addresses in the execution record are always 64-bit (zero-extended if needed), but only the first AddrWidth bytes are significant.
9	pad[3]		

8 System calls table

System calls are taken care of in the **Execution** table. There is however some data pertaining to a system call as a whole (or to both corresponding "entry" and "exit" execution records), namely the "name" and arguments (count and values), among others, which are included in the **System calls** table.

8.1 Name and argument count

Internally, *Windows* system calls are made through a common entry point, passing a numerical ID as the kernel function identifier, which is then dispatched via a branch table. Branch tables are OS version-specific but corresponding kernel functions are also exported by name. The latter is more consistent across various *Windows* versions since those functions are part of the Native API. Nevertheless, from a version to another, some functions are added, removed or have their argument changed. The argument count is somewhat exported as well, although there is no reliable ways to automatically know how many arguments have the calls on 64-bit *Windows* with 4 or less arguments.

For the sake of trace analysis, function name and argument count (where possible) from all supported OSes' branch tables (see ***supported OSes.docx***), were automatically extracted and merged into a unified table. Functions with different name or argument count were manually given different unique IDs. The result is found in ***syscall_unified_id_v0.h***, provided with this documentation. This is version 0 of the unified list, the first and only for now. Obviously, as newer OS versions are added in the future, this list is likely to change, and new versions will be generated. Most enumerated names are constructed on the following pattern:

SYSCALL_ExportedName_MMmm_N

Where **MM** and **mm** are respectively major and minor version of the earliest *Windows* (among supported versions) to provide this function, and **N** is the argument count. For 64-bit OS calls with 4 or less arguments, argument count is assumed to be equal to the 32-bit version argument count, which is known. Should there not be a 32-bit equivalent, **N** is written as **x64**. Note that **N** might be wrong in some cases, Native API being mostly undocumented. Some functions with a different argument count may also have the same exported name, even on the same *Windows* version. Other bits of text were added to a few names to resolve conflicts between OS versions.

A trace analysis tool that does system call specific analysis should rely on that uniform ID in each system call record (and the developer can rely on the enumeration name), although the original call ID is also provided as a convenience.

8.2 Argument values

Argument values are available from registers and/or the stack, depending on the system call convention used, and found in the few execution records preceding the call. To make things easier for the analysis tool, argument values are properly extracted and copied into a simple array in system call records.

8.3 Extras

Several system calls use kernel objects handles as arguments. Those objects are opaque to user-mode code, although in several cases the Win32 native API provides way to query them. Such queries must be done online, at trace time. For now, UMTracer records such extra information for a handful of functions and UMTIndex forwards it to system call records. However, this is experimental code, format for those extras is far from definitive and its description is beyond the scope of this document. See section What is in an execution record for more information and minimal workarounds.

8.4 File format

`System call` table is a collection of variable-size records, split into two files.

`syscall.vtable`: packed collection of variable-size system call records.
Size: variable

Offset	Type	Name	Description
`0`	`i32`	`Version`	Format version number (only version 0 is currently supported)
`4`	`i32`	`UnifiedIDsVer`	Uniform ID enum version (only version 0 is currently supported). See section Name and argument count above.
`8`	`i64`	`RecordCount`	Valid IDs for both files are in [0, `RecordCount` [
`Offset[i]`	`syscallRec`		Variable-length record (see `syscallRec` structure below) for each `i` in [0, `RecordCount` [. Offset and size are given in ***`syscall.offsets`***. Records in the file are *not* sorted by their ID.

`syscall.offsets`: Absolute file offsets and size for each record in ***`syscall.vtable`***.
Size: `RecordCount` * 16

Offset	Type	Name	Description
`0`	`i64Pair[RecordCount]`	`(Offset, Size)`	Each pair holds the absolute file offset and size of the corresponding record in ***`syscall.vtable`***.

syscallRec: fixed-size part of an instruction record
Size: 64
Note: All data pointed to (via file pointers) by this record is stored consecutively so that the whole record (including pointed data) fits in a single block of size **Size[i]**.

Offset	Type	Name	Description
0	i16	RawID	OS version-specific function ordinal. Relevant only when combined with the OS version the trace was created on.
2	i16	UniformID	OS version-independent kernel function ID. See section Name and argument count above.
4	i8	Convention	System call convention used to make that call: • 0: Unknown • 1: 32-bit fast system call (syscall/sysenter) • 2: 64-bit fast system call (syscall/sysenter) • 3: alternate system call (int 0x2e on Windows) • 4: WOW64 system call • 5: software interrupt (int instruction, except alternate system call)
5	bool	Completed	1 if the system call returned like a normal function (EndEID is valid), 0 otherwise. In the latter case, either the call never completed before the end of the thread or trace (EndEID = −1) or its exit was shortcut by any kind of context jump (EndEID is valid). See system call related execution in section Execution record format.
6	i8	ArgCount	Number of arguments in array pointed to by Args. Always 4 for 64-bit calls with an unknown argument count between 0 and 4.
7	i8	ExtraCount	Number of "extra" items in array pointed to by ExtrasFP. Undocumented, see section Extras above.
8	i64	BeginEID	System call entry execution record ID.
16	i64	EndEID	System call exit execution record ID. It is −1 if the thread was still in the context of the call when it stopped (or where the tracer stopped recording). Otherwise, it is either a normal (CF-wise) system call execution record, or a skipped system call exit (Completed = 0, see system call related execution in section Execution record format).
24	i64	BeforeSP	Value of stack pointer (esp/rsp) before the call. Zero-extended for 32-bit traces.
32	i64	Result	Return value in eax/rax at exit point, zero-extended for 32-bit traces. Invalid (garbage) if Completed = 0.
40	i64[ArgCount] *	ArgsFP	FP on the array of ArgCount argument values. FP is relative to the start of this syscallRec record. Arguments of 32-bit calls are zero-extended to 64 bits. Arguments are in left-to-right order of exported kernel function prototypes.
48	buffer	ExtrasFP	FP on the array of ExtraCount "extras" items. FP is relative to the start of this syscallRec record. Undocumented, see section Extras above.

9 Module table

Modules (.exe and .dll) content is not stored directly in the trace. It rather appears in the trace as it is used, either through memory references in executed instructions, or instruction code bytes themselves. Optionally, UMTracer can dump in the trace all files mapped into memory (this is how modules are typically imported into the process). Module initial contents will then appear as a simple memory block dumped along with the registers context on system call exit execution record. This is not complete since some modules are already mapped into memory when the tracer starts, and this can be rather inefficient. For example, in a file explorer window, the same few MBs in size DLL can be mapped and unmapped for each visible file.

The `Module` table is stored in ***`module.itable`***. Its header follows the fixed-size table format described in section Fixed-size records, where `record` type is `moduleRec`, described below. It also has the following table-specific fields in its header:

Offset	Type	Name	Description
32	`buf[ExtraSize] *`	`ExtraFP`	Absolute file pointer to a buffer containing collections of strings pointed to by module records.
40	`i64`	`ExtraSize`	Total size of the buffer pointed to by `ExtraFP`.

`moduleRec`: fixed-size part of an instruction record
Size: 48

Offset	Type	Name	Description
0	`i64`	`LoadEID`	Load time, expressed through the approximate ID of the execution record about to be recorded at that point.
8	`i64`	`UnloadEID`	Unload time, expressed through the approximate ID of the execution record about to be recorded at that point. Set to `-1` if the trace recording ends before module is unloaded.
16	`i64`	`StartAddr`	Memory address where the module was loaded
24	`i64`	`EndAddr`	Memory address of the last module byte (inclusive)
32	`uStr *`	`WinNameFP`	FP to full path and file name, in usual Windows format, such as `C:\Windows\System32\mf.dll` FP is relative to start of file (absolute FP)
40	`uStr *`	`DeviceNameFP`	FP to Low-level device path, such as `\Device\HarddiskVolume3\Windows\System32\mf.dll` FP is relative to start of file (absolute FP)

10 Thread table

The **Thread** table is stored in ***thread.itable***. Its header follows the fixed-size table format described in section Fixed-size records, where **record** type is **threadRec**, described below.

threadRec:
Size: 48

Offset	Type	Name	Description
0	i32	ID	Self-ID in the current table.
4	i32	WinTID	Windows thread ID while it was running.
8	i64	TIBAddr	Address of Thread Information Block. TIB contents is recorded along with registers context in “thread begin” execution records (see section Execution record format).
16	i64	FirstEID	First execution record for that thread, normally of type “thread begin”.
24	i64	LastEID	Last execution record for that thread, normally of type “thread end”, or 1 if the tracer was interrupted while the thread was still running.
32	i64	RecCount	Number of execution records that belong to that thread.

11 Context map

11.1 Registers

Some execution records contain a registers context dump. All register values are stored in a contiguous buffer. The **Context map** defines its data layout, i.e. where each register is given a register name. Although the current version of UMTracer does not generate mixed 32-bit and 64-bit code traces, the **Context map** supports both of them at once. It also supports registers that may not exist on the CPU the trace was made, although those are marked as "not present".

The **Context map** also provides the largest enclosing register for each register, both *absolute* and *effective*. On one hand, an *absolute* largest enclosing register is taken within all *known* register (from all possible names in the enum, see section Intel XED), even though it may not exist on the CPU the trace was made. For instance, the largest register enclosing **al** is **rax**, even on 32-bit traces. Noteworthy enclosures are:

- **rax/rbx/rcx/rdx** ⊃ **ah**/**bh**/**ch**/**dh**
- x87 registers (80-bit) ⊃ MMX registers (64-bit), but in reverse order (**st0** ⊃ **mm7**, **st1** ⊃ **mm6**, … **st7** ⊃ **mm0**)
- YMM registers (256-bit) ⊃ XMM registers (128-bit)

On the other hand, *effective* enclosing registers deal only with valid registers on the original CPU and according to the actual bitness (32- or 64-bit mode) of an instruction. In all cases, the largest enclosing register for special values (**STACKPUSH**, **MEM0**, etc.) is the register itself.

WARNING: *absolute* largest enclosing registers depend on the actual XED register table version in use (see section Intel XED). This is typically the latest XED library bundled with Pin for x86-x64, at the time UMTIndex was built, unless there was no significant change since the version in use. For instance, ZMM registers (from Intel AVX-512 instructions introduced in Intel Knights Landing microarchitecture) are expected to make their way into x86-x64 processors around 2015.

11.2 File format

The file format has been designed for quick lookups, without the need for bounds checking. Per register information is indexed by register name. There are about 175 registers name for now, and several others are expected to be added within a few years. Therefore, all arrays have a nice and easy size of 512 elements and unused elements are filled with valid "no register here" data. Also, all register names are represented using 16-bit integers.

context_map.itable: context dump data layout and inclusion relations among registers
Size: 17424

Offset	Type	Name	Description
0	ui16	DumpSize	Size of a context dump buffer. All offsets in RegMap32 and RegMap64 are smaller than DumpSize.
2	ui16	ElemCount	Count of valid elements in all four arrays below. All Arrays have 512 elements, but only the first ElemCount are meaningful.
4	pad[12]		
16	i16[512]	Largest	*Absolute* largest enclosing register for each register (see Registers above)
1040	ac[16][512]	Name	Array of 512 "strings", the textual name of each register. Each string is actually a fixed 16-bytes array and names are null-terminated.
9232	regInfo[512]	RegMap32	Per register contextual information for instructions in 32-bit mode code segment.
13328	regInfo[512]	RegMap64	Per register contextual information for instructions in 64-bit mode (a.k.a. *long mode)* code segment.

regInfo: contextual information about a register (32/64-bit dependant)
Size: 8

Offset	Type	Name	Description
0	ui16	Offset	Offset of the register in a context dump buffer, relative to the start of the buffer, or 0xffff if the current register is not part of this 32/64-bit context.
2	ui8	Size	Size of the register in a context dump buffer, or 0 if the current register is not part of this 32/64-bit context.
3	pad[1]		
4	i16	EffLargest	*Effective* largest enclosing register for current register (see Registers above)
6	pad[2]		

12 Address space usage

This table contains all memory addresses referenced by the trace, either in instruction operands or context dumps. Memory addresses are rounded to 4 kB page granularity, and consecutive pages are grouped into page ranges. Therefore, this table is likely a superset of addresses that are really accessed.

The **Address space usage** table is stored in ***address_space.itable***. Its header follows the fixed-size table format described in section Fixed-size records, where **record** type is **addressRange**, described below.

addressRange: page-granularity address range
Size: 16

Offset	Type	Name	Description
0	i64	First	First byte of a page range, always a multiple of 4 kB.
8	i64	Last	Last byte (included) of a page range. Always greater than **First** and one less than a multiple of 4 kB.

13 Trace.xml

Trace.xml is meant to contain small amounts of heterogeneous data, as well as trace meta-data. It uses plain XML, no specific format or schema being specified for now. Element path names make them somewhat self-descriptive, and besides a few exceptions below, are not described in this document.

File contents is expected to change in the future, although a few elements are most likely to remain because of their potential use by a trace analyzer tool:

- **`/Xed/Version`**: *Intel XED* version used to generate the trace. Tells which external *XED* tables to use (see section Intel XED). *XED* version is actually derived from the version string supplied by *XED* library, but this could change in the future. Currently, only version 20130904 is supported.
    - An analyzer should not try to process a trace made with a newer *XED* version since most constants from *XED* tables are likely to be wrong.
- **`/UMTracer/Exec/SyscallTable/Version`**: Uniform system call IDs table version. Only version 0 exists for now. See section Name and argument count.
- **`/App/ProcessID`**: Can be useful when analysing some system calls.

14 External tables

14.1 Intel *XED*

UMTracer and UMTIndex both use *Intel XED*, a powerful instruction decoding/encoding library, either directly or through Intel *Pin*. `Instruction` table provides detailed enough instruction decoding so that *XED* ***library*** (or linking against any C library at all) should not be required to analyze a trace. Nevertheless, some fields refer to lengthy *XED* enumerations tables (register name, instruction mnemonic, etc.) with self-descriptive value names. Those are required for a trace analyzer tool developer to make sense out of those IDs.

While it would be possible to translate them into home-made enums, it was not required for now. It might be in the future, should *XED* versioning (*XED* instruction set support is always up to date with *upcoming* Intel processors) become problematic. Relevant header files from *XED* are found in ***`XED v20130904`*** folder along with this document. Readily usable in C/C++, their contents can be easily adapted for other languages such as Java or Python.

- ***`xed-reg-enum.h`***: register "name" enum. Some values are not likely to find their way in a user-mode trace (`cr`$_x$ or `dr`$_x$ registers), but most others are used. In particular, `XED_REG_INVALID` is used in trace records to mean "no register", "empty operand slot", etc.
- ***`xed-iclass-enum.h`***: instruction mnemonics
- ***`xed-iform-enum.h`***: specific form of an instruction, such as `add reg,reg` and `add mem,reg`.
- ***`xed-icategory-enum.h`***: instruction categories. Sometimes provides easier analysis dispatching than a lengthy switch-case on form or mnemonic.

Although not directly used by trace records, ***`idata.txt`***, found along with other *XED* files, provides the complete mapping between instruction-related enums. For more information about *XED*, see the *Pin* page: http://www.pintool.org/downloads.html.

14.2 System call IDs

As explained in section Name and argument count, system call identifiers (their ordinal in kernel function tables) are OS version-specific. The tracer converts them to a uniform ID so that the same ID relates to the same exported kernel function (as long as the function semantics and argument count remains the same across OS versions). That enum table can be found in ***`syscall_unified_id_v0.h`***. Enum names are described in section Name and argument count as well.