

# DIOTA

v0.91

# Manual

8/6/04





# Table of Contents

1	Introduction.....	1
2	Quickstart for the impatient.....	2
3	Building DIOTA.....	3
	Prerequisites.....	3
	Getting the source.....	3
	Building.....	3
	Compile-time options.....	3
4	Running DIOTA.....	5
	Run-time parameters.....	6
5	Description of backends.....	8
	none.....	9
	trace-mem.....	10
	trace-code & trace-code-bin.....	12
	count-code.....	15
	diota-leaks.....	16
	diota-dr.....	18
	diota-record & diota-replay.....	20
	diota-stealth.....	22
6	Using DIOTA in combination with GDB.....	23
7	Writing your own backend.....	24
8	Client controlled instrumentation.....	30
9	History of DIOTA.....	31
10	Bibliography.....	32
	Journal papers.....	32
	Conference papers.....	32
	Other publications.....	32

## Index of Tables

Table 1 - List of variables offered by DIOTA.....	24
Table 2 - List of macros offered by DIOTA.....	24
Table 3 - List of functions offered by DIOTA.....	25
Table 4 - List of interceptable events.....	27
Table 5 - List of #THREAD_EVENT# types.....	28
Table 6 - List of functions available to the traced application.....	30

# 1 Introduction

DIOTA (Dynamic Instrumentation, Optimisation and Transformation of Applications) is a tool for instrumenting IA32 binaries running on GNU/Linux. The tool correctly deals with programs that contain traditionally hard to instrument features such as data in code and code in data. The technique does not require reverse engineering, program understanding tools or heuristics about the compiler or linker used. The basic idea is that instrumented code is generated on the fly, while the original process is used for data accesses. DIOTA comes with a number of useful backends to check programs for faulty memory accesses, data races, deadlocks,... and perform basic tracing operations, e.g. tracing all memory accesses, all code being executed, to perform coverage analysis, ...

At this moment, DIOTA can:

- add instrumentation to all memory operations in an executable and the libraries it uses (either statically or dynamically linked)
- call a user-provided function for each executed basic block
- analyse and/or modify system call in- and output
- intercept calls to dynamically linked procedures (e.g. malloc(), printf(), ...)

DIOTA has been implemented as a dynamic loadable library that can be attached to arbitrary Linux programs (using a loader to attach itself to statically linked applications). DIOTA supports

- dynamically and statically linked applications
- multithreaded applications (both the old LinuxThreads as the new Native Posix Threads Library (NPTL) are supported)
- Position Independent Executables (PIE)
- signal handlers and exceptions
- procedures registered with atexit()
- `_init` and `_fini` sections in the application itself, but not the `_init` and `_fini` sections of (all) dynamically linked libraries
- self-modifying code
- all IA32 instructions, including MMX, SSE, ...

As DIOTA needs no special information, applications can be compiled and linked in the usual way. DIOTA doesn't need symbol or debug information, but this information can be used by backends for showing call stacks, etc.

## 2 Quickstart for the impatient

```
[~/work/diota] autoconf # only needed if you use a CVS version
[~/work/diota] ./configure
[~/work/diota] make
[~/work/diota] . alias
[~/work/diota] cd /bin
[/bin] trace-mem # trace all memory operations
[/bin] uname # this application will be traced
[/bin] head -20 /tmp/diota.log-uname-<your-login-name>
```

The last command will show you the first 20 lines of the generated log file, containing a list of all memory operations:

```
==== This is DIOTA v0.91, compiled by ronsse@... on Tue Jun 29 17:40:02 CEST 2004
==== CMD: uname
==== MEM_INIT: Trace all data accesses =====
T P ip address len pre -post opcode opcode_bytes
==== DIOTA_init2: start
==== Instrumenting: MEM
L 0 4001a608 bffff574 4 bffff5c4 leave c9
L 0 4001a609 bffff578 4 4000bf71 ret c3
L 0 4000befd 4001907c 4 00000000 mov 0x7c(%esi),%edx 8b 56 7c
L 0 4000bec0 bffff5a0 4 00000002 mov 0xfffffdc(%ebp),%edi 8b 7d dc
X 0 4000bec3 bffff5a0 4 00000002-00000001 decl 0xfffffdc(%ebp) ff 4d dc
L 0 4000be70 bffff5b4 4 400164d0 mov 0xfffff0(%ebp),%eax 8b 45 f0
L 0 4000be73 bffff5a0 4 00000001 mov 0xfffffdc(%ebp),%edi 8b 7d dc
L 0 4000be76 400166a8 4 40019d10 mov 0xd8(%eax),%ecx 8b 88 d8 01 00 00
L 0 4000be7c 40019d14 4 40016b28 mov (%ecx,%edi,4),%esi 8b 34 b9
L 0 4000be7f 40016c9c 1 15 movzbl 0x174(%esi),%eax 0f b6 86 74 01 00 00
L 0 4000be8c 40016b2c 4 40016af8 mov 0x4(%esi),%eax 8b 46 04
S 0 4000be92 40016c9c 1 15 -ld mov %dl,0x174(%esi) 88 96 74 01 00 00
L 0 4000be98 40016af8 1 2f [/] movzbl (%eax),%ecx 0f b6 08
L 0 4000bea3 40016b70 4 00000000 mov 0x48(%esi),%edx 8b 56 48
[/bin] none # disable diota
```

## 3 Building DIOTA

### Prerequisites

Apart from the obvious prerequisites such as `gcc`, `ld`, `make`, ... DIOTA needs a fairly recent `binutils` package. Running `diota` requires sourcing a file, and as this file contains `bash` commands, you should use `bash` as your shell.

### Getting the source

The source can be downloaded from <http://www.elis.UGent.be/diota/download/>. Nightly builds can be found in <http://www.elis.UGent.be/diota/download/unstable/>.

### Building

If you are using a CVS version (the CVS repository is not world readable), start with:

```
autoconf
```

If you are using a `.tar.bz2` version or a CVS version:

```
./configure
make      # Please ignore the warnings...
```

### Compile-time options

Both DIOTA and a number of backends accept compile-time options. They can be set using the `O` `make` parameter, e.g.

```
make O=-Doption
```

will add `-Doption` to each compile command. Use quotes if you want to set multiple options, e.g.

```
make O="-Doption1 -Doption2".
```

DIOTA accepts a number of options, but only a limited number are to be used by an end user:

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
DEBUG	If defined, the output file generated will contain a lot of debug information. Only useful if you want to debug DIOTA or a backend.		
EBUG	Same as above, allows you to use <code>-DEBUG</code> instead of <code>-DDEBUG</code> , which is much more fun...		
SYSIO_UNBUFFERED	Output is normally buffered, which means that you lose the last part if DIOTA crashes. Setting this option disables buffering, at the expense of a larger slowdown.		Slows down tracing tremendously.
STATIC_HANDLERS	Activates a slower callback mechanism. You need this if an application uses the <code>diota_pause_mem_instr()</code> and <code>diota_resume_mem_instr()</code> functionality (see page 30).		

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
STACK_GAP	DIOTA reserves STACK_GAP bytes on the stack before calling itself. Useful if an application writes below the stack pointer, e.g. set to 1024 for the Jikes RVM.	0	Automatically sets STATIC_HANDLERS.

The options for the backends will be listed with their description.

## 4 Running DIOTA

To use diota, type:

```
. alias
```

This defines an alias for each available backend. You will get something like

```
The following aliases have been defined: bandwidth, bb, callchain,
deadlock, diota-ca, diota-dr, diota-leaks, diota-malloc, diota-mem,
diota-record, diota-replay, diota-stealth, dummy, opt,
read_trace_code_bin, selfmod, trace-code-bin, trace-code, trace-mem,
wait,
```

In order to use a backend, just type the name of the backend and then execute your dynamically linked program, e.g.

```
trace-mem
./path/application
```

A log file will be created as

```
/tmp/diota.log-._path_application-username
```

The name of the log file is composed of four parts:

1. /tmp. This is the default directory for log files, the directory can be changed by setting the DIOTAPARAM environment variables (see page 6).
2. diota.log-
3. the name of the application. The directory separator / is replaced with \_
4. your user name

Additionally, thread specific log files are created as

```
/tmp/diota.log-._path_application-username.T0
/tmp/diota.log-._path_application-username.T1
...
```

### Example<sup>1</sup>

```
[~/diota/test] gcc -o mem4 mem4.c -g
[~/diota/test] ./mem4
Segmentation fault
[~/diota/test] diota-malloc      # check for invalid memory accesses
[~/diota/test] ./mem4
Segmentation fault
[~/diota/test] cat /tmp/diota.log-._mem4-ronsse
===== This is DIOTA v0.91, compiled by ronsse@... on Mon Jun 28 17:19:00 CEST 2004
===== CMD: ./mem4
===== DIOTA-MALLOC: check memory accesses + malloc(),... =====
===== No initialization check =====
===== DIOTA_init2: start
===== Instrumenting: MEM

=====Error:Invalid Memory access @ 0x1000 of size 4 using instruction @ 0x8048356!

Address 0x1000 isn't allocated
The instruction is:
        movl    $0x1, (%eax)                c7 00 01 00 00 00

The place where the error occurred is:
        [0x08048356]: main, /user/ronsse/work/diota/test/mem4.c:10
        9:          int * a=0x1000;
```

---

<sup>1</sup> In order to obtain the same results: `export DIOTAPARAM=malloc_check_initialized=0,show_source=1`

```
10:     a[0]=1;      // -
11: }
```

The call chain is:

```
====End Error
signal 11: passing through to instrumented handler
```

Diota only attaches itself to applications that can be found (or a file with the same name) in the current directory, e.g.

```
trace-mem
ls
```

will probably not trace `ls` (unless you execute this command in `/bin`), but if you do a `touch ls` first, `ls` will be traced.

Diota can also instrument statically linked binaries if you put `diota` in front of the the actual command, e.g.

```
diota ./statically-linked-application
```

For statically linked binaries, a small piece of the diota loader is also traced; the actual trace starts a few lines after

```
==== restarting from ....
```

in the log.

You can use different backends at the same time, e.g.

```
trace-mem
trace-code
./application
```

In this case, the log files will contain information from both backends.

Type `none` to unset the list of backends to be used, e.g.

```
trace-mem
./application1
trace-code
./application2
none
trace-code
./application3
```

will execute

- application1 with the trace-mem backend
- application2 with the trace-mem and the trace-code backend
- application3 with the trace-code backend

## ***Run-time parameters***

DIOTA (and its backends) accept a number of parameters to turn options on and off. Parameters are set using the `DIOTAPARAM` environment variable using `parameter=value` pairs, separated by commas. Value is either an integer or a string. The strings `on`, `yes` and `true` are silently replaced<sup>2</sup> by the integer 1, while `off`, `no` and `false` are replaced by 0. Therefore, the following lines have the same meaning:

```
export DIOTAPARAM=a=0,b=yes
export DIOTAPARAM=a=No,b=TRUE
export DIOTAPARAM=a=false,b=1
```

---

<sup>2</sup> String replacement is case-insensitive, e.g. you can use `true`, `True` or `TRUE`.

DIOTA accepts the following parameters:

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
show_source	If set, and if the application was compiled with debug support (-g), certain backends will write source code to the log file.	0	Causes slowdown.
log_dir	Directory where the log files will be created.	/tmp	
log_param	String to be attached to end of names of log files.		Must be a string that can't be parsed as an integer, e.g. use “_4” instead of “4”.
only_local_dir	If unset, DIOTA will attach itself to all applications, not only those in the current directory.	1	
gdb	Automatically attaches GDB to the traced application. See page 23 for information on how to use GDB.	0	
smc	Turns on support for self-modifying code.	0	Causes considerable slowdown

The parameters for the backends will be listed with their description.

## 5 Description of backends

All backends are implemented as dynamic loadable libraries. They are loaded together with an application by setting the `LD_PRELOAD` environment variable. However, you don't have to change this variable by hand: the file `SO/alias` defines a number of aliases. To source this file, type `. alias` (that is: `<dot><space>alias`).

The most important backends will be described briefly in the following pages.

## **none**

### **Purpose**

This is not an actual backend: none simply clears the LD\_PRELOAD environment variable.

### **Example**

```
[ronsse@SD90MAC ~] echo $LD_PRELOAD
[ronsse@SD90MAC ~] trace-mem
[ronsse@SD90MAC ~] echo $LD_PRELOAD
libdiota_libc.so:/user/ronsse/work/diota/SO/trace-mem.so:
[ronsse@SD90MAC ~] trace-code
[ronsse@SD90MAC ~] echo $LD_PRELOAD
libdiota_libc.so:/user/ronsse/work/diota/SO/trace-code.so:libdiota_libc.so:/user/ronsse/work/diota/SO/trace-
mem.so:
[ronsse@SD90MAC ~] none
[ronsse@SD90MAC ~] echo $LD_PRELOAD
[ronsse@SD90MAC ~] trace-code
[ronsse@SD90MAC ~] echo $LD_PRELOAD
libdiota_libc.so:/user/ronsse/work/diota/SO/trace-code.so:
```

## trace-mem

### Purpose

Creates a trace of all memory operations. The log file will contain one<sup>3</sup> line for each executed memory operation: <T> <P> <ip> <address> <len> <pre>-<post> <opcode> <opcode\_bytes> with

- <T>: the type of memory operation. This can be
  - L: a load operation
  - S: a store operation
  - X: a modify operation: a load followed by a store to the same address
- <P>: the thread executing the operation, is 0 for sequential programs.
- <ip>: the address of the memory instruction.
- <address>: the address accessed by the operation.
- <len>: the number of bytes accessed by the operation. This is the number of bytes read or written, except for modify operations, where the numbers of bytes accessed is twice this value.
- <pre>: the value before the operation, only shown if len<=4. Also shows an ASCII character if len=1 and the character is printable.
- <post>: only shown for store or modify operations: the value after the operation. Same rules as above apply.
- <opcode>: a string representation of the instruction.
- <opcode\_bytes>: a byte representation of the instruction.

### Usage

Load the backend with `trace-mem`.

### Runtime parameters

None

### Example

```
[ronsse@SD90MAC /tmp] trace-mem
[ronsse@SD90MAC /tmp] touch uname
[ronsse@SD90MAC /tmp] uname
Linux
[ronsse@SD90MAC /tmp] ls -l diota.log-uname-ronsse*
-rw----- 1 ronsse users 318 jun 29 12:25 diota.log-uname-ronsse.T0
-rw----- 1 ronsse users 2341695 jun 29 12:25 diota.log-uname-ronsse
[ronsse@SD90MAC /tmp] head -20 diota.log-uname-ronsse
===== This is DIOTA v0.91, compiled by ronsse@... on Mon Jun 28 17:19:00 CEST 2004
===== CMD: uname
===== MEM_INIT: Trace all data accesses =====
T P ip address len pre -post opcode opcode_bytes
===== DIOTA_init2: start
===== Instrumenting: MEM
L 0 4001a608 bffff3e4 4 bffff434 leave c9
L 0 4001a609 bffff3e8 4 4000bf71 ret c3
L 0 4000befd 4001907c 4 00000000 mov 0x7c(%esi),%edx 8b 56 7c
L 0 4000bec0 bffff410 4 00000002 mov 0xffffffffc(%ebp),%edi 8b 7d dc
X 0 4000bec3 bffff410 4 00000002-00000001 decl 0xffffffffc(%ebp) ff 4d dc
L 0 4000be70 bffff424 4 400164d0 mov 0xfffffffff0(%ebp),%eax 8b 45 f0
L 0 4000be73 bffff410 4 00000001 mov 0xffffffffc(%ebp),%edi 8b 7d dc
```

<sup>3</sup> Some instructions generate two lines, e.g. a memory-indirect call accesses memory twice: reading the target address and storing the return address on the stack.

L 0 4000be76	400166a8	4	40019d10	mov	0x1d8(%eax),%ecx	8b 88 d8 01 00 00
L 0 4000be7c	40019d14	4	40016b40	mov	(%ecx,%edi,4),%esi	8b 34 b9
L 0 4000be7f	40016cb4	1	15	movzbl	0x174(%esi),%eax	0f b6 86 74 01 00 00
L 0 4000be8c	40016b44	4	40016b10	mov	0x4(%esi),%eax	8b 46 04
S 0 4000be92	40016cb4	1	15	mov	%dl,0x174(%esi)	88 96 74 01 00 00
L 0 4000be98	40016b10	1	2f	movzbl	(%eax),%ecx	0f b6 08
L 0 4000bea3	40016b88	4	00000000	mov	0x48(%esi),%edx	8b 56 48

## **trace-code & trace-code-bin**

### **Purpose**

Both backends create a trace of all executed instructions. While the log file created by `trace-code` contains readable information, `trace-code-bin` works much faster and creates a much smaller binary file (`/tmp/diota.trace-code-bin-<application>-<username>`). This binary file has to be postprocessed with `dump_trace_code_bin`, which creates the same readable information that can be found in the log file created by `trace-code`. Information is written in basic blocks that start with an instruction that was the target of a jump, and stops with an indirect jump. If the same piece of code is executed a number of times, e.g. in a loop, the information is only written once. The log file contains two types of lines:

1. `C <P> <count> <start> - <stop>`with
  - `<P>` the thread that executed the instructions
  - `<count>` the number of times these instructions were executed in a loop
  - `<start>` the address of the first instruction
  - `<stop>` the address of the last instruction
2. `> <address>: <instruction> <opcode bytes>`. These lines are generated after a line starting with `C`, one line for each instruction with
  - `<address>` the address of the instruction
  - `<instruction>` a string representation of the instruction
  - `<opcode_bytes>` the opcode bytes

The backend is accompanied by two scripts that postprocess this information:

- `show_source` generates the sequence of all executed instruction, annotated with the corresponding source. This is of course only possible for code that contains debug information by using the `-g` flag when compiling. This command should be executed in the directory that contains the executable.
- `profile_code` generates a profile of all instructions.

### **Usage**

Load the backend with `trace-code` or `trace-code-bin`. If `trace-code-bin` was used, postprocess with `dump_trace_code_bin /tmp/diota.trace-code-bin-...` to get readable output. If necessary, postprocess the readable log file with `show_source /tmp/diota.log-...` or `profile_code /tmp/diota.log-...`

### **Runtime parameters**

None

### **Example**

```
[ronsse@SD90MAC ~/work/diota] cd /bin
[ronsse@SD90MAC /bin] trace-code
[ronsse@SD90MAC /bin] uname
Linux
[ronsse@SD90MAC /bin] cat /tmp/diota.log-username-ronsse
...
C 0 1 400a5290 - 400a52c3
> 400a5290: movzwl (%edi),%edx          0f b7 17
> 400a5293: mov    %eax,%esi                 89 c6
> 400a5295: mov    (%eax),%eax               8b 00
> 400a5297: subl  $0x2,0xffffffff0(%ebp)    83 6d f0 02
> 400a529b: add   $0x2,%edi                  83 c7 02
> 400a529e: test  %eax,%eax                  85 c0
```

```

> 400a52a0: mov    %edx,0xffffffff(%ebp)      89 55 ec
> 400a52a3: je     0x400a5280                  74 db
> 400a52a5: mov    %eax,%edx                  89 c2
> 400a52a7: mov    %esi,%esi                  89 f6
> 400a52a9: lea   0x0(%edi),%edi              8d bc 27 00 00 00 00
> 400a52b0: movzwl (%edx),%ecx                0f b7 0a
> 400a52b3: cmp   %cx,0xffffffff(%ebp)       66 39 4d ec
> 400a52b7: je     0x400a52c6                  74 0d
> 400a52b9: add   $0x4,%esi                   83 c6 04
> 400a52bc: mov   (%esi),%ecx                 8b 0e
> 400a52be: test  %ecx,%ecx                   85 c9
> 400a52c0: mov   %ecx,%edx                   89 ca
> 400a52c2: jne   0x400a52b0                   75 ec
C 0 46 400a52b0 - 400a52c3 (executed 46 times)
> 400a52b0: movzwl (%edx),%ecx                0f b7 0a
> 400a52b3: cmp   %cx,0xffffffff(%ebp)       66 39 4d ec
> 400a52b7: je     0x400a52c6                  74 0d
> 400a52b9: add   $0x4,%esi                   83 c6 04
> 400a52bc: mov   (%esi),%ecx                 8b 0e
> 400a52be: test  %ecx,%ecx                   85 c9
> 400a52c0: mov   %ecx,%edx                   89 ca
> 400a52c2: jne   0x400a52b0                   75 ec
C 0 1 400a52b0 - 400a52b8
> 400a52b0: movzwl (%edx),%ecx                0f b7 0a
> 400a52b3: cmp   %cx,0xffffffff(%ebp)       66 39 4d ec
> 400a52b7: je     0x400a52c6                  74 0d
C 0 1 400a52c6 - 400a52dd
> 400a52c6: mov   %edi,0x4(%esp)              89 7c 24 04
> 400a52ca: mov   0xffffffff(%ebp),%ecx       8b 4d f0
> 400a52cd: mov   %ecx,0x8(%esp)              89 4c 24 08
> 400a52d1: mov   (%esi),%edx                 8b 16
> 400a52d3: add   $0x2,%edx                   83 c2 02
> 400a52d6: mov   %edx,(%esp)                 89 14 24
> 400a52d9: call  0x400e2960                   e8 82 d6 03 00
C 0 1 400e2960 - 400e2982
> 400e2960: push  %ebp                        55
> 400e2961: xor   %eax,%eax                   31 c0
> 400e2963: mov   %esp,%ebp                   89 e5
> 400e2965: push  %edi                        57
...

```

```

[ronsse@SD90MAC /bin] trace-code-bin
[ronsse@SD90MAC /bin] uname
logging binary trace to /tmp/diota.trace-code-bin-uname-ronsse
Linux
[ronsse@SD90MAC /bin] dump_trace_code_bin /tmp/diota.trace-code-bin-uname-ronsse | head
C 0 1 4001a69c - 4001a69d
> 4001a69c: leave                                c9
> 4001a69d: ret                                  c3
C 0 1 4000bf71 - 4000bf72
> 4000bf71: jmp   0x4000befd                     eb 8a
C 0 1 4000befd - 4000bf03
> 4000befd: mov   0x7c(%esi),%edx                8b 56 7c
> 4000bf00: test  %edx,%edx                      85 d2
> 4000bf02: je    0x4000bec0                     74 bc
C 0 1 4000bec0 - 4000bec9

```

```

[ronsse@SD90MAC ~/work/diota/test] show_source /tmp/diota.log-._source_example-ronsse
...
40090ad1: call  *0x8(%ebp)                    |
0804833c: push  %ebp                          | source_example.c:3 <main>      |main(){
0804833d: mov   %esp,%ebp                      |
0804833f: sub   $0x8,%esp                      |
08048342: and   $0xfffffffff0,%esp            |
08048345: mov   $0x0,%eax                      |
0804834a: sub   %eax,%esp                      |
0804834c: movl  $0x7,0xffffffff(%ebp)         | source_example.c:4 <main>      | int i=7;
08048353: lea  0xffffffff(%ebp),%eax          | source_example.c:5 <main>      | i++;
08048356: incl  (%eax)                          |
08048358: mov   0xffffffff(%ebp),%edx         | source_example.c:6 <main>      | i*=6;
0804835b: mov   %edx,%eax                      |
0804835d: shl  %eax                             |
0804835f: add  %edx,%eax                       |
08048361: shl  %eax                             |
08048363: mov   %eax,0xffffffff(%ebp)         |

```

```

08048366: leave          | source_example.c:7 <main>      |}
08048367: ret                |                                |
...
[ronsse@SD90MAC ~/work/diota/test] profile_code /tmp/diota.log-._profile_code_example-ronsse
...
  1 > 4000c195: test    %edi,%edi                85 ff
  1 > 4000c197: je      0x4000c26b            0f 84 ce 00 00 00
 10 > 4000c19d: movl   $0x1,0xffffffffe8(%ebp)  c7 45 e8 01 00 00 00
 10 > 4000c1a4: mov    0xffffffffec(%ebp),%eax  8b 45 ec
 10 > 4000c1a7: cmp    %edi,0x4(%eax)          39 78 04
 10 > 4000c1aa: je     0x4000c1be            74 12
  9 > 4000c1ac: lea   0x0(%esi),%esi          8d 74 26 00
 43 > 4000c1b0: incl  0xffffffffe8(%ebp)      ff 45 e8
 43 > 4000c1b3: mov   0xffffffffec(%ebp),%ecx  8b 4d ec
 43 > 4000c1b6: mov   0xffffffffe8(%ebp),%eax  8b 45 e8
 43 > 4000c1b9: cmp   %edi,(%ecx,%eax,4)      39 3c 81
 43 > 4000c1bc: jne   0x4000c1b0            75 f2
 10 > 4000c1be: mov   0xffffffffe8(%ebp),%esi  8b 75 e8
 10 > 4000c1c1: inc   %esi                    46
 10 > 4000c1c2: cmp   %edx,%esi              39 d6
 10 > 4000c1c4: mov   %esi,0xfffffffffb(%ebp)  89 75 bc
 10 > 4000c1c7: jae   0x4000c260            0f 83 93 00 00 00
  9 > 4000c1cd: mov   0xffffffffe8(%ebp),%edx  8b 55 e8
  9 > 4000c1d0: mov   0xffffffffec(%ebp),%ecx  8b 4d ec
...

```

## count-code

### Purpose

This backend gathers some statistical data about the executed code:

- number of instructions executed
- number of instruction bytes read
- average number of instructions per basic block
- average instruction length

### Usage

Load the backend with `count-code`.

### Compile time options

None

### Runtime parameters

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
<code>count_code_interval</code>	The interval (in basic blocks) to decide when to print out statistical information.	100000	

### Example

```
[ronsse@SD90MAC /bin] count-code
[ronsse@SD90MAC /bin] DIOTAPARAM=count_code_interval=10000 ./pwd
/bin
[ronsse@SD90MAC /bin] cat /tmp/diota.log-._pwd-ronsse
===== This is DIOTA v0.91, compiled by ronsse@... on Wed Jul 14 13:07:22 CEST 2004
===== CMD: ./pwd
===== CWD: /bin
===== COUNT-CODE: Gather statistics about code accesses, dumping every 10000 basic blocks =====
===== DIOTA_init2: start
===== Instrumenting: CODE
10000 basic blocks elapsed: instructions executed: 39571, instruction bytes read: 112041,
instructions/basic block: 3.957, bytes/instruction: 2.831
===== DIOTA_finil
===== Backend fini():s
12058 basic blocks elapsed: instructions executed: 47912, instruction bytes read: 136873,
instructions/basic block: 3.973, bytes/instruction: 2.856
[ronsse@SD90MAC /bin] DIOTAPARAM=count_code_interval=2500 ./pwd
/bin
[ronsse@SD90MAC /bin] cat /tmp/diota.log-._pwd-ronsse
===== This is DIOTA v0.91, compiled by ronsse@... on Wed Jul 14 13:07:22 CEST 2004
===== CMD: ./pwd
===== CWD: /bin
===== COUNT-CODE: Gather statistics about code accesses, dumping every 2500 basic blocks =====
===== DIOTA_init2: start
===== Instrumenting: CODE
2500 basic blocks elapsed: instructions executed: 10479, instruction bytes read: 31892,
instructions/basic block: 4.191, bytes/instruction: 3.043
5000 basic blocks elapsed: instructions executed: 20159, instruction bytes read: 60246,
instructions/basic block: 4.031, bytes/instruction: 2.988
7500 basic blocks elapsed: instructions executed: 29863, instruction bytes read: 86041,
instructions/basic block: 3.981, bytes/instruction: 2.881
10000 basic blocks elapsed: instructions executed: 39571, instruction bytes read: 112041,
instructions/basic block: 3.957, bytes/instruction: 2.831
===== DIOTA_finil
===== Backend fini():s
12058 basic blocks elapsed: instructions executed: 47912, instruction bytes read: 136873,
instructions/basic block: 3.973, bytes/instruction: 2.856
```

## diota-leaks

### Purpose

This backend checks an execution for memory leaks and tries to pinpoint exactly where memory gets lost (as opposed to just where the memory is allocated).

### Usage

Load the backend with `diota-leaks`.

### Compile time options

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
POINTER_CHECK	Report when loading pointers to already freed blocks (even if in the mean time already a new block has been allocated at the same place)		More slowdown.

### Runtime parameters

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
leaks_print_interval	The interval to decide when to print out the leaks is decided by counting the number of allocations. If this number is larger than 10000 x leaks_print_interval leak information is written. Default is 10000 allocations. Set to -1 to only write out all leaks grouped at the end.	1	
leaks_track	Diota-leaks writes the block id of the first leaked block of every group of leaks. If you set this parameter to such a number and rerun the program, all operations on this block are written to the log file as well (all places in the program where the reference count of this block increases and decreases). If set, leaks_print_interval is automatically set to -1 to avoid interleaving of this info and detected leaks.		

### Example

```
[~/work/diota/test] cat leaks_example.c
#include <stdlib.h>

int main(void) {
    void * a, *b;
    a=malloc(321);
    b = a;
    a=malloc(456);
    b = NULL;
    return 0;
}
[~/work/diota/test] gcc leaks_example.c -o leaks_example -g
[~/work/diota/test] export DIOTAPARAM=show_source=1
[~/work/diota/test] diota-leaks
[~/work/diota/test] cat /tmp/diota.log-._leaks_example-ronsse
...
===== Flushing diota-leaks errors collected until now:

*** Warning, losing (due to stack shrinking) last reference to a block of memory (first leaked
block id = 2, reference at 0xbffff504, block at 0x804b5e0, ip = 0x80483b8, 1 occurrence(s)) in
thread 0 at:
    [0x080483b8]: main(), /user/ronsse/work/diota/test/leaks_example.c:12
                11:     return 0;
```

```
12:-> } <-
13:   } [0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

The last reference to that block we know of was created at 0x80483a9:

```
[0x080483a9]: main(), /user/ronsse/work/diota/test/leaks_example.c:9
8:   b = a;
9:-> a=malloc(456); <-
10:   b = NULL;
[0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

This block was allocated at 0x80483a1:

```
[0x080483a1]: main(), /user/ronsse/work/diota/test/leaks_example.c:9
8:   b = a;
9:-> a=malloc(456); <-
10:   b = NULL;
[0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

\*\*\* Warning, losing (due to stack shrinking) last reference to a block of memory (first leaked block id = 1, reference at 0xbffff500, block at 0x804b498, ip = 0x80483b8, 1 occurrence(s)) in thread 0 at:

```
[0x080483b8]: main(), /user/ronsse/work/diota/test/leaks_example.c:12
11:   return 0;
12:-> } <-
13:   } [0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

The last reference to that block we know of was created at 0x8048396:

```
[0x08048396]: main(), /user/ronsse/work/diota/test/leaks_example.c:8
7:   a=malloc(321);
8:-> b = a; <-
9:   a=malloc(456);
[0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

This block was allocated at 0x8048388:

```
[0x08048388]: main(), /user/ronsse/work/diota/test/leaks_example.c:7
6:
7:-> a=malloc(321); <-
8:   b = a;
[0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

-----

The following memory regions were dynamically allocated, but they were never released:

[0x0804b498 -> 0x0804b5d8 : 321 bytes]

```
[0x08048388]: main(), /user/ronsse/work/diota/test/leaks_example.c:7
6:
7:-> a=malloc(321); <-
8:   b = a;
[0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

[0x0804b5e0 -> 0x0804b7a7 : 456 bytes]

```
[0x080483a1]: main(), /user/ronsse/work/diota/test/leaks_example.c:9
8:   b = a;
9:-> a=malloc(456); <-
10:   b = NULL;
[0x40145acf]: __libc_start_main+223, /lib/tls/libc.so.6
```

## diota-dr

### Purpose

This backend detects data races that occur during a execution. Data races can occur in a parallel programs, but also in sequential programs that use signal handlers. The detector searches for conflicting memory accesses (load/store, store/load or store/store) that (can) occur in parallel. Pieces of code that do not satisfy the 'happened-before relation' are considered parallel. Due to the fact that detecting races uses huge amounts of memory and time, the diota-dr backend uses two phases:

1. During a first execution, the addresses of all memory operations are collected and parallel acceses are compared. For each found data race, some information is written to the file `/tmp/diota.dr-application-user`.
2. During a second execution, more information about the first found data race is collected: call stack, the actual instruction, ... Please note that the second execution should exhibit the same behaviour as the first execution, the diota-record/diota-replay backend can help you to accomplish this.

### Usage

#### Runtime parameters

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
<code>dr_combine_accesses</code>	If set to 1, diota will combine memory accesses. Will speed up data race detection.	0	Warning: all backends will get the combined accesses! This option is silently ignored during the second execution.
<code>dr_no_stack_accesses</code>	If set to 1, stack accesses will not be checked for data races. Only works in combination with <code>dr_combine_accesses</code> , and will introduce an additional speedup.	0	Setting this to 1 automatically sets <code>dr_combine_accesses</code> also to 1. This option is silently ignored during the second execution

### Example

```
[~/work/diota/backend_dr/test/par] cat race_example.c

#include <pthread.h>

unsigned global;

void * doit2(void * arg) {
    switch((int)arg){
        case 1: global+=4; break;
        case 2: global+=5; break;
    }
    return NULL;
}

void * doit(void * arg) {
    return doit2(arg);
}

int main(){
    pthread_t t1, t2;

    pthread_create(&t1, NULL /*attr*/, doit, (void*)1 /*arg*/);
    pthread_create(&t2, NULL /*attr*/, doit, (void*)2 /*arg*/);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("global=%d\n", global);

    return 0;
}
```

```

}

[~/work/diota/backend_dr/test/par] cc race_example.c -o race_example -g -lpthread

[~/work/diota/backend_dr/test/par] ./race_example

Datarace detected between thread 2 (load, store, or modify) and thread 1 (store) at address
0x080496ec
Thread 2: segment 1, current vector clock: { T0:3 T1:0 T2:1 } (current thread 2)
Thread 1: segment 1, current vector clock: { T0:2 T1:1 T2:0 } (current thread 2)

global=9

----
One or more data races have been detected.
Information about the data races was written to /tmp/diota.dr-._race_example-ronsse.
Rerun to obtain more information about the first data race in that file.

[~/work/diota/backend_dr/test/par] cat /tmp/diota.dr-._race_example-ronsse
T2 S1..1 ? & T1 S1..1 s @ 0x80496ec

[~/work/diota/backend_dr/test/par] ./race_example
Searching for information about previously found data race involving 0x80496ec.
-----
Offending instruction of thread 1:
0x0804844e: addl    $0x4,0x80496ec                83 05 ec 96 04 08 04

[0x0804844e]: doit2(), /user/ronsse/work/diota/backend_dr/test/par/race_example.c:8
7:      switch((int)arg){
8:->    case 1: global+=4; break; <-
9:      case 2: global+=5; break;

Call stack of thread 1:
[0x08048471]: doit(), /user/ronsse/work/diota/backend_dr/test/par/race_example.c:15
14:    void * doit(void * arg){
15:->  return doit2(arg); <-
16:    }

[0x405f58a3]: diota_pthread_start+187, /user/ronsse/work/diota/SO/libdiota.so
[0x405f5990]: no source code or symbol information found

[0x404af987]: no source code or symbol information found
-----
Offending instruction of thread 2:
0x08048457: addl    $0x5,0x80496ec                83 05 ec 96 04 08 05

[0x08048457]: doit2(), /user/ronsse/work/diota/backend_dr/test/par/race_example.c:9
8:      case 1: global+=4; break;
9:->    case 2: global+=5; break; <-
10:    }

Call stack of thread 2:
[0x08048471]: doit(), /user/ronsse/work/diota/backend_dr/test/par/race_example.c:15
14:    void * doit(void * arg){
15:->  return doit2(arg); <-
16:    }

[0x405f58a3]: diota_pthread_start+187, /user/ronsse/work/diota/SO/libdiota.so
[0x405f5990]: no source code or symbol information found

[0x404af987]: no source code or symbol information found
-----
global=9

```

## ***diota-record & diota-replay***

### **Purpose**

As parallel programs can be non deterministic (consecutive runs with the same input can result in different executions) it is rather difficult to use cyclic debugging techniques (repeating the same execution over and over while zooming in on the bug) in order to debug such a program. In order to be able to use those techniques we need a tool that traces information about an execution so it can be replayed for debugging. Diota-record and diota-replay form such a tool: diota-record records the order of synchronization operations (trace file=/tmp/diota.record-...) and diota-replay replays them.

### **Usage**

For tracing an execution: load the backend with `diota-record`. For replaying a traced execution: first unload `diota-record` with `none`, and then load `diota-replay`.

### **Example**

```
[ronsse@SD90MAC ~/work/diota/test] cat ptest.c

#include <pthread.h>
#include <semaphore.h>

pthread_mutex_t mutex;
int global=0;

doit(int arg){
    int count=100;

    while (count-->0) {
        pthread_mutex_lock(&mutex);
        switch (arg){
            case 1: global+=1000; break;
            case 2: global/=2; break;
        }
        pthread_mutex_unlock(&mutex);
        while (random()>100000) // introduce non determinism
            ;
    }
    pthread_exit(NULL);
}

main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, (void (*)(void *))doit, (void *)1);
    pthread_create(&t2, NULL, (void (*)(void *))doit, (void *)2);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("global=%d\n", global);
}
[ronsse@SD90MAC ~/work/diota/test] gcc ptest.c -o ptest -lpthread
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=0
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=94000
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=96000
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=13000
[ronsse@SD90MAC ~/work/diota/test] diota-record
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=0
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=14000
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=25036
[ronsse@SD90MAC ~/work/diota/test] none
[ronsse@SD90MAC ~/work/diota/test] diota-replay
```

```
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=25036
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=25036
[ronsse@SD90MAC ~/work/diota/test] ./ptest
global=25036
[ronsse@SD90MAC ~/work/diota/test] ls -l /tmp/diota.record-._ptest-ronsse
-rw----- 1 ronsse users 112 jul 14 18:17 _._ptest.t
```

## diota-stealth

### Purpose

Diota-stealth is a backend that tries to hide diota and its backends from the application, e.g. the application will be unaware of the fact that diota opens trace files. As such, diota-stealth is almost always used in combination with other backends.

### Usage

### Runtime parameters

<i>name</i>	<i>purpose</i>	<i>default</i>	<i>comment</i>
stealth_unlink	If set to 0, the unlink system call will not be executed.	1	
stealth_closeonexec	If set to 0, the FD_CLOEXEC bit will be ignored for the fcntl system call.	1	

### Example 1

```
[~/work/diota/backends/diota-stealth/test] cat stealth_example.c

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main(){
    printf("LD_PRELOAD=%s\n", getenv("LD_PRELOAD"));
    printf("file opened as %d\n", open("/etc/motd", O_RDONLY));
}

[~/work/diota/backends/diota-stealth/test] ./stealth_example
LD_PRELOAD=
file opened as 3

[~/work/diota/backends/diota-stealth/test] trace-mem

[~/work/diota/backends/diota-stealth/test] ./stealth_example
LD_PRELOAD=libdiota_libc.so:/user/ronsse/work/diota/SO/trace-mem.so:
file opened as 5

[~/work/diota/backends/diota-stealth/test] diota-stealth

[~/work/diota/backends/diota-stealth/test] ./stealth_example
LD_PRELOAD=(null)
file opened as 3
```

### Example 2

```
[/tmp] touch rm # make sure rm is traced
[/tmp] touch a; rm a; ls a
ls: a: No such file or directory
[/tmp] diota-stealth
[/tmp] touch a; rm a; ls a
ls: a: No such file or directory
[/tmp] export DIOTAPARAM=stealth_unlink=no
[/tmp] touch a; rm a; ls a
a
```

## 6 Using DIOTA in combination with GDB

If you want to use `gdb` together with `diota`, define set the `DIOTAPARAM` parameter `gdb` to 1 and use `diota` in the normal way; `diota` will start `gdb` for you and attach it to the running program

```
[ronsse@SD90MAC ~/work/diota/test] export DIOTAPARAM=gdb=1
[ronsse@SD90MAC ~/work/diota/test] trace-mem
[ronsse@SD90MAC ~/work/diota/test] ./test
diota: pid 6022, sleeping 3 seconds
GNU gdb Red Hat Linux (5.3.90-0.20030710.41rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are welcome to change it and/or distribute copies of it under
certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"... (no debugging
symbols found)...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

Attaching to program: /user/ronsse/work/diota/test/test, process 6022
<<cut>>
(gdb)
```

Type ``c'` to continue the execution in `gdb`.

Afterwards, you will almost immediately encounter two `SIGSEGV`'s, and you may encounter more later on. This is normal, and you can ignore them (by typing ``c'` again). The cause is the fact that every program contains at least one "call 0x0" instruction, so `DIOTA` causes this exception when it tries to find out which instructions lie at its target address. As long as the exceptions occur inside the function "`diota_asm_getlong()`", they will be caught and handled correctly.

Note that you cannot set breakpoints in your code, as an instrumented copy is executed instead of the original program. You still can view the contents of variables, however. You can also translate an instrumentation address to an original program address by typing

```
p/x diota_translation_lookup_prog_address(<instrumentation_address>)
```

## 7 Writing your own backend

As this chapter will demonstrate, writing your own backend is very simple. The smallest possible backend is:

```
#include "diota.h"

void function(void){
}

INIT(function)
```

In order to build this backend (called backend.c), place it in the backends/ directory and add the name of the backend to the BACKENDS target in backends/Makefile, e.g.

```
../SO/backend.so
```

After compiling diota in the usual way and resourcing the alias file with

```
. alias
```

you can enable the backend by typing the name of it:

```
backend
```

Of course, this backend doesn't perform anything useful. In order to communicate with the diota instrumentation engine, diota offers a number of variables (Table 1), macros (Table 2) and functions (Table 3)

<i>Variable</i>	<i>Remark</i>
int diota_tty	File descriptor used to access the main log file.

*Table 1 - List of variables offered by DIOTA.*

<i>Macro</i>	<i>Remark</i>
INIT(void(*start)(void))	Bootstrap macro: the argument is the function that will be executed when the backend is loaded.
THREADID	The thread identifier.
THREAD_TTY	File descriptor used to access the thread specific log files.

*Table 2 - List of macros offered by DIOTA.*

<i>Function</i>	<i>Remark</i>
sysio_open sysio_close sysio_read sysio_write sysio_exit sysio_popen sysio_pclose sysio_putchar sysio_strlen sysio_sprintf sysio_vsprintf sysio_printf sysio_vprintf	As it is unsafe to use the normal I/O functions of the C-library, diota uses its own <sup>4</sup> versions. The functions behave identical to the functions open(), close(), ... with one exception: the third argument (mode_t mode) for open() is mandatory.
sysio_fputs sysio_fprintf sysio_vfprintf	Identical to fputs(), fprintf(), ... but the first argument is a file descriptor (int) instead of a file pointer (FILE *).
void sysio_flush(int fd)	Flushes file fd, does nothing if SYSIO_UNBUFFERED is defined.
diota_malloc diota_calloc diota_realloc diota_free diota_mmap diota_munmap	Replacement functions for malloc(), free(), mmap() and munmap().
int diota_lock(int * lock) int diota_unlock(int * lock)	Functions to acquire/release a spin lock.
unsigned diota__interlocked_increment(unsigned *) unsigned diota__interlocked_decrement(unsigned *)	Atomically increments/decrements an integer and returns the new value.
void diota_yield(void)	Executes the sys_sched_yield system call.
char * diota_program_name(void)	Returns the name of the application that is being traced.
char * diota_user_name(void)	Returns the name of the user executing the application.
char * diota_create_log_name(char * info)	Returns a file name that can be used by a backend as a log file. File name contains the argument (info), the name of the application and the user name. Honours the log_dir parameter.
int diota_exec_stat_linked(void)	Returns TRUE if application is statically linked, FALSE otherwise.
unsigned diota_callback(...)	The most important function: registers callbacks with diota. Table 4 lists the interceptable events.

*Table 3 - List of functions offered by DIOTA.*

The `diota_callback` function enables a backend to register callbacks with diota. This function is typically used by the bootstrap function of the backend.<sup>5</sup> The prototype of the function is

<sup>4</sup> DIOTA uses a modified version of the sysio library of Frank Cornelis.

<sup>5</sup> Actually, this is the only place where it is safe to register callbacks.

```
unsigned diota_callback(char *function, unsigned default_address, unsigned callback)
```

with

- `function`: the name of the function or the event we want to intercept. All events are strings that start with a # in order to distinguish them from real functions.
- `default_address`: the default address of the function, not used for events.
- `callback`: the address of the function that should be called when the function is intercepted or the event occurs.

The function returns the address of the intercepted function, or 0 for events. Table 4 lists all interceptable events. A small example on how to use `diota_callback`:

```
#include "diota.h"

static void * (*real_malloc)(size_t);

static void * our_malloc(size_t size){
    void * result = real_malloc(size);
    sysio_fprintf(diota_tty, "malloc(%d) @ 0x%x\n", size, result);
    return result;
}

static void memory_access(const t_mem_access *ma){
    sysio_fprintf(diota_tty, "%.8x/%d/%d/%.8x %x\n", ma->ip, ma->type, ma->len, ma->address, ma->esp);
}

static void _malloc_backend_init(){
    real_malloc = diota_callback("malloc", (unsigned)malloc, (unsigned)our_malloc);
    diota_callback("#DATA#", 0, (unsigned)memory_access);
}

INIT(_malloc_backend_init)
```

<i>event</i>	<i>when called</i>	<i>arguments of callback</i>
#START#	<b>After</b> all backends have been initialised but <b>before</b> the execution starts.	None.
#STOP#	<b>After</b> the execution, with the log files still open.	None.
#POSTPROCESS#	<b>After</b> the execution, with the log files closed.	<b>char *</b> : name of the main log file
#DATA#	<b>Before</b> each memory access.	<b>const t_mem_access *</b> : pointer to struct with information about the memory access. Fields are: <ul style="list-style-type: none"> <li>• <b>unsigned ip</b>: address of instruction</li> <li>• <b>unsigned type</b>: 0 (load), 1 (store) or 2 (modify)</li> <li>• <b>unsigned len</b>: number of bytes accessed</li> <li>• <b>unsigned address</b>: address of accessed memory</li> <li>• <b>unsigned esp</b>: the stack pointer</li> </ul>
#CODE#	<b>Before</b> each executed basic block.	<b>unsigned</b> : address of first instruction <b>unsigned</b> : address of last instruction
#SIGNAL#	<b>Before</b> and <b>after</b> the execution of a signal handler.	<b>int</b> : signal number before, and 0 after the execution of the signal handler. <b>unsigned</b> : pid of sender (if applicable)
#THREAD_EVENT#	At thread-related events.	See Table 5 for a list of event types.
#SYSCALL_MODIFIER#	<b>Before</b> a system call.	<b>pusha_registers_t*</b> : pointer to structure with contents of register. Fields are: <ul style="list-style-type: none"> <li>• <b>unsigned edi</b></li> <li>• <b>unsigned esi</b></li> <li>• <b>unsigned ebp</b></li> <li>• <b>unsigned esp</b></li> <li>• <b>unsigned ebx</b></li> <li>• <b>unsigned edx</b></li> <li>• <b>unsigned ecx</b></li> <li>• <b>unsigned eax</b></li> </ul>
#SYSCALL_ANALYZER#	<b>After</b> a system call.	<b>pusha_registers_t</b> : structure (see above) with contents of register before the system call <b>unsigned *</b> : pointer to system call result (=0%eax)
<functionname>	<b>Instead of</b> <functionname>	The actual arguments for the function.

Table 4 - List of interceptable events

The #THREAD\_EVENT# callback is called for all thread related events, an argument depicts the event type. This callback provides more functionality than can be obtained by simply intercepting pthread\_create(), pthread\_exit and pthread\_join(). The event types are shown in Table 5.

<i>type</i>	<i>when</i>	<i>executor</i>	<i>argument</i>	<i>comment</i>
THREAD_EVENT_CREATE	<b>Before</b> the creation of a new thread.	The parent thread.		The value (void*) returned by the callback is forwarded to the child using THREAD_EVENT_START
THREAD_EVENT_CREATED	<b>After</b> the creation of a new thread.	The parent thread.	The ID of the new thread.	
THREAD_EVENT_START	<b>Before</b> the new thread starts executing.	The child thread.	The value (void*) returned by THREAD_EVENT_CREATE	
THREAD_EVENT_EXIT	<b>After</b> a thread exits.	The child thread.		Is called always, even if a thread exits without using pthread_exit()
THREAD_EVENT_JOIN	<b>After</b> a thread was joined.	The parent thread.	The ID of the thread that exited.	

Table 5 - List of #THREAD\_EVENT# types.

Please note that THREAD\_EVENT\_START and THREAD\_EVENT\_EXIT are not used for the main thread; use the #START# and #STOP# callback instead. The following example and figure shows a typical usage for #THREAD\_EVENT#.

```
#include <diota.h>

static void * thread_event(unsigned event, void * arg){
    void * result=NULL;
    switch (event){
        case THREAD_EVENT_CREATE:    result=thread_create();           break;
        case THREAD_EVENT_CREATED:    thread_created((unsigned) arg);  break;
        case THREAD_EVENT_START:      thread_start(arg);               break;
    }
    return result;
}

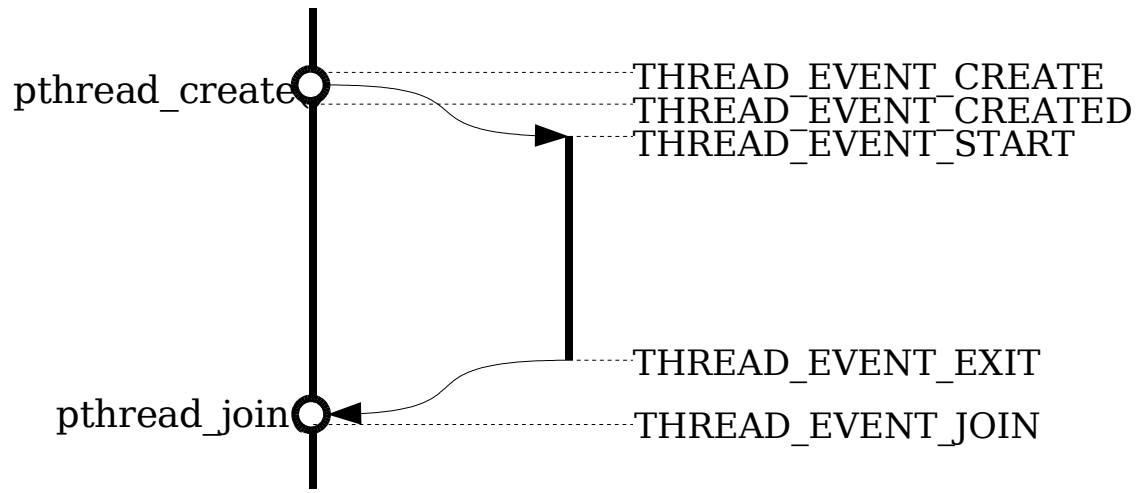
static void * thread_create(){
    sysio_printf("Thread %d will create a new thread\n", THREADID);
    return 666;
}

static void thread_created(unsigned tid){
    sysio_printf("Thread %d has created thread %d\n", THREADID, tid);
}

static void thread_start(void * param){
    sysio_printf("I'm new thread %d, and I received the parameter %d\n", THREADID, (unsigned)param);
}

static void backend_init(){
    diota_callback("#THREAD_EVENT#", 0, (unsigned)thread_event);
}

INIT(backend_init)
```



## 8 Client controlled instrumentation

Diota allows the client application that is being executed to influence the instrumentation. Table 6 describes the functions that can be used by the client. In order to use these functions the application should include

```
#include "client/include/diota_client.h
```

and the following link arguments should be used:

```
-Lclient/lib -rpath client/lib -ldiota_client
```

<i>Function</i>	<i>Remark</i>
<code>void diota_pause_mem_instr(void)</code>	Stop calling memory instrumentation callbacks.
<code>void diota_resume_mem_instr(void)</code>	Resume calling memory instrumentation callbacks.
<code>void diota_pause_code_instr(void)</code>	Stop calling code instrumentation callbacks.
<code>void diota_resume_code_instr(void)</code>	Resume calling code instrumentation callbacks.
<code>void diota_thread_reset_instr(void)</code>	Reset the generated instrumentation code of the current thread. Useful if you want to support self-modifying code without incurring the huge overhead of the automatic detection of self-modifying code (only available if you set <code>smc=1</code> in <code>DIOTAPARAM</code> ). Logfiles are not emptied, but a marker is added to note that the instrumentation was reset.
<code>void diota_mainlog(char * string)</code>	Write a string to the main log file.
<code>void diota_threadlog(char * string)</code>	Write a string to the thread specific log file.

*Table 6 - List of functions available to the traced application.*

## 9 History of DIOTA

It is impossible to talk about DIOTA without mentioning its predecessor JiTI. JiTI (Just in Time Instrumentation) was developed by Michiel Ronsse (Ghent University, Belgium) in 1997 for his PhD about data race detection. For his PhD, he needed an instrumentation tool, and at that time the available tools had several shortcomings. The basic concept of JiTI was *the clone*: the complete memory image of a running application was cloned in the address space of the application and this clone was instrumented: individual instructions were replaced with jumps to instrumentated code. As all SPARC instructions have the same length, this was fairly easy to implement. JiTI could implement the complete application but not the dynamically linked libraries.

In 2000, a port of JiTI to the Intel platform running the GNU/Linux operating system was started, using the name JiTI86. This turned out to be a rather complex undertaking, caused by the numerous 'challenges' offered by the Intel architecture. In September 2001, Jonas Maebe (Ghent University, Belgium) started his Masters Thesis about JiTI86 and he joined the porting: he added support for instrumenting dynamically linked libraries, made JiTI86 thread safe, made JiTI86 much faster and added support for self modifying code. In contrast to JiTI for the SPARC, JiTI86 was not implemented as one big dynamic loadable library, but consisted of an instrumentation engine accompanied by a number of backends that implemented the actual logic of different tools (e.g. data race detection). As such, the name JiTI86 was not completely accurate anymore and the new name DIOTA (Dynamic Instrumentation, Optimization and Transformation of Applications) was chosen. Diota is an ancient Roman word that means 'a vase or drinking cup having two handles or ears' and we thought it perfectly describes our tool: DIOTA uses two handles, one handle to access the unmodified data and one handle to execute the modified code.

In 2003, Bastiaan Stougie (Delft University, The Netherlands) started his Masters Thesis on optimized data race detection. The basic idea was that memory accesses could be combined (both in place and in time) and less (expensive) callbacks to the data race backend were necessary, speeding the detection up. In order to do this an extensive rewrite of the instrumentation engine was necessary and control flow analysis was added.

## 10 Bibliography

Papers about DIOTA or JiTI, the predecessor of DIOTA for Solaris/SPARC:

### **Journal papers**

1. Ronsse, M.; De Bosschere, K. *RecPlay: A Fully Integrated Practical Record/Replay System*. ACM Transactions on Computer Systems. Vol. 17 (2). 1999. pp. 133-152

### **Conference papers**

1. Maebe, J.; Ronsse, M.; De Bosschere, K. *Precise Detection of Memory Leaks*. Proceedings of the Second International Workshop on Dynamic Analysis (WODA 2004). 2004. pp. 25-31
2. Maebe, J.; De Bosschere, K. *Instrumenting self-modifying code*. Proceedings of the Fifth International Workshop on Automated Debugging: AADEBUG2003. 2003. pp. 103-113
3. Maebe, J.; Ronsse, M.; De Bosschere, K. *DIOTA: Dynamic Instrumentation, Optimization and Transformation of Applications*. Compendium of Workshops and Tutorials Held in conjunction with PACT'02: International Conference on Parallel Architectures and Compilation Techniques. 2002.
4. Ronsse, M.; De Bosschere, K. *JiTI: A Robust Just in Time Instrumentation Technique*. Proceedings of WBT-2000 (Workshop on Binary Translation). 2000. pp. 1-12
5. Ronsse, M.; De Bosschere, K. *JiTI: Tracing Memory References for Data Race Detection*. Parallel Computing: Fundamentals, Applications and New Directions. North Holland. Advances in Parallel Computing. Vol. 12. 1998. pp. 327-334

### **Other publications**

1. Maebe, J.; Ronsse, M.; De Bosschere, K. *Self-modifying code: instrumentation challenges*. Fourth FTW PhD Symposium. 2003. On CD
2. Stougie, B. *Optimization of a Data Race Detector*. Afstudeerwerk TU-Delft. Promotor: De Bosschere, K. 2003.
3. Maebe, J. *Dynamische optimalisatie van programma's*. Masters thesis, FTW, RUG. Promotor: De Bosschere, K. 2002.
4. Ronsse, M. *JiTI: Just in Time Instrumentation*. ELIS Technical Report. 1997. pp. 1-32

## Alphabetical Index

#CODE#	27	diota-record	20
#DATA#	27	diota-replay	20
#POSTPROCESS#	27	diota-stealth	22
#SIGNAL#	27	DIOTAPARAM	6
#START#	27	dr_combine_accesses	18
#STOP#	27	dr_no_stack_accesses	18
#SYSCALL_ANALYZER#	27	<b>E</b>	
#SYSCALL_MODIFIER#	27	EBUG	3
#THREAD_EVENT#	27	<b>G</b>	
<b>C</b>		gdb	7
count_code_interval	15	<b>I</b>	
count-code	15	INIT	24
<b>D</b>		<b>L</b>	
DEBUG	3	leaks_print_interval	16
diota__interlocked_decrement	25	leaks_track	16
diota__interlocked_increment	25	log_dir	7
diota_callback	25	log_param	7
diota_calloc	25	<b>M</b>	
diota_create_log_name	25	malloc_check_initialized	5
diota_exec_stat_linked	25	<b>N</b>	
diota_free	25	none	9
diota_lock	25	<b>O</b>	
diota_mainlog	30	only_local_dir	7
diota_malloc	25	<b>P</b>	
diota_mmap	25	POINTER_CHECK	16
diota_munmap	25	<b>S</b>	
diota_pause_code_instr	30	show_source	5, 7
diota_pause_mem_instr	3, 30	smc	7, 30
diota_program_name	25	STACK_GAP	4
diota_realloc	25	STATIC_HANDLERS	3
diota_resume_code_instr	30	stealth_closeonexec	22
diota_resume_mem_instr	3, 30	stealth_unlink	22
diota_thread_reset_instr	30	sysio_close	25
diota_threadlog	30	sysio_exit	25
diota_translation_lookup_prog_address	23	sysio_flush	25
diota_tty	24	sysio_fprintf	25
diota_unlock	25	sysio_fputs	25
diota_user_name	25	sysio_open	25
diota_yield	25	sysio_pclose	25
diota-dr	18	sysio_popen	25
diota-leaks	16		

sysio_printf	25	THREAD_EVENT_CREATE	27
sysio_putchar	25	THREAD_EVENT_CREATED	28
sysio_read	25	THREAD_EVENT_EXIT	28
sysio_sprintf	25	THREAD_EVENT_JOIN	28
sysio_strlen	25	THREAD_EVENT_START	28
SYSIO_UNBUFFERED	3	THREAD_TTY	24
sysio_vfprintf	25	THREADID	24
sysio_vprintf	25	trace-code	12
sysio_vsprintf	25	trace-code-bin	12
sysio_write	25	trace-mem	10

## **T**