Minemu: The World's Fastest Taint Tracker

Erik Bosman, Asia Slowinska, and Herbert Bos

Vrije Universiteit Amsterdam

Abstract. Dynamic taint analysis is a powerful technique to detect memory corruption attacks. However, with typical overheads of an order of magnitude, current implementations are not suitable for most production systems. The research question we address in this paper is whether the slow-down is a fundamental speed barrier, or an artifact of bolting information flow tracking on emulators really not designed for it. In other words, we designed a new type of emulator from scratch with the goal of removing superfluous instructions to propagate taint. The results are very promising. The emulator, known as *Minemu*, incurs a slowdown of 1.5x-3x for real and complex applications and 2.4x for SPEC INT2006, while tracking taint at byte level granularity. *Minemu*'s performance is significantly better than that of existing systems, despite the fact that we have not applied some of their optimizations yet. We believe that the new design may be suitable for certain classes of applications in production systems.

Keywords: dynamic taint tracking, JIT compilation, intrusion detection

1 Introduction

Fifteen years after Aleph One's introduction to memory corruption [17], and despite a plethora of counter-measures (like ASLR [3], PaX/DEP [18], and canaries [7]), buffer overflows alone rank third in the CWE SANS top 25 most dangerous software errors[1]. Dynamic taint analysis (DTA) [16, 6] is very effective at stopping most memory corruption attacks that divert a program's control flow. Moreover, the wealth of information it collects about untrusted data makes it well-suited for forensics and signature generation [26]. Unfortunately, software DTA is so slow that in practice its use is limited to non-production machines like honeypots or malware analysis engines.

In this paper, we describe *Minemu*, a new emulator architecture that speeds up dynamic taint analysis by an order of magnitude compared to well-known taint systems like taint-check [16], Vigilante [6], and Argos [20]. Specifically, *Minemu* brings down the slowdown due to taint analysis to 1.5x-3x for real applications. Unless your application really starves for performance, a slowdown of, say, 2x to be safe from most memory corruption attacks might be a reasonable price for many security-sensitive systems.

Current counter measures do not stop memory corruption. Typical memory corruption attacks overwrite a critical value in memory to divert a program's flow of control to code injected or selected by the attacker. We argue that current protection mechanisms (like PAX/DEP, ASLR, and canaries) are insufficient. Consider for instance, the buffer

[1] Version 2.0, 2010 `http://www.sans.org/top25-software-errors/`

underrun vulnerability in Figure 1. The example is from a Web server request parsing procedure in `nginx-0.6.32` [1]—in terms of market share across the million busiest sites, the third largest Web server in the world[2], hosting about 23 million domains worldwide at the time of writing. The buffer underrun allows attackers to execute arbitrary programs on the system. They do not trample over canaries. They do not execute code in the data segment. Since they call into libc, they are not stopped by ASLR either.

In reality, the situation is worse. All defense mechanisms used in practice, including the three above, have weaknesses that allow attackers to circumvent them, and/or situations in which they cannot be applied (e.g., JIT code *requires* data pages to be executable). Moreover, a recent report indicates that many programs either do not use features like DEP or ASLR at all, or use them incorrectly [25]. Finally, legacy binaries often cannot even be protected using such measures.

Dynamic Taint Analysis (DTA) is one of the few techniques that protect legacy binaries against all memory corruption attacks on control data. Because of its accuracy, the technique is very popular in the systems and security community—witness a string of publications in the last few years in tier-1 venues, including SOSP [6], CCS [30], NDSS [16], ISCA [9], MICRO [8], EUROSYS [20], ASPLOS [28], USENIX [5, 12], USENIX Security [29], Security& Privacy [24], and OSDI [13]—it is clearly well liked.

Frustratingly though, DTA is too slow to be used in production systems. In practice, its use is limited to non-production machines like honeypots or malware analysis engines. With slow-downs that often exceed an order of magnitude, few are keen to apply taint analysis to, say, their webserver or browser.

Contributions The research question we address in this paper is whether the slowdown is a fundamental performance barrier, or an artifact of bolting information flow tracking on emulators not designed for it? To answer this question, we designed a new emulator architecture for the `x86` architecture from scratch—with the sole purpose of minimizing the instructions needed to propagate taint. The emulator, *Minemu*, reduces the slowdown of DTA in most real applications to a factor of 1.5 to 3. It is significantly faster than existing solutions, even though we have not applied some of their most significant optimizations yet. We believe that the new design may be suitable for certain classes of applications in production systems.

Specifically, what we did not do is rely on static analysis. In principle, it is possible to improve performance by means of statically analyzing the program to determine which instructions need taint tracking and which do not. Unfortunately, static analysis and even static disassembly of stripped binaries is still an unsolved problem. Therefore, the authors of the best-known work in this category [23], assume the presence of at least some symbolic information (like the entry points of functions). In practice, this is typically not available. In fact, we do not even check at (dynamic) translation time whether the data is tainted (whether we could follow a fast path) as proposed by the authors of LIFT [22]. In LIFT terminology, *Minemu* always takes the slow path. As a result, *Minemu*'s performance is independent of the amount of taint in the inputs.

[2] `http://news.netcraft.com/archives/2011/03/09/march-2011-web-server-survey.html#more-3991`

We show that, despite not using these optimization techniques and using pure dynamic translation, *Minemu*'s performance exceeds that of even the fastest existing systems [23, 22, 14].

The first key observation underlying *Minemu* is that fast DTA requires a fast emulator. Thus, we designed a new and highly efficient `x86` emulator from scratch. Compared to other emulators like QEMU [2], *Minemu* translates much larger blocks in one go. Additionally, the emulator applies caching aggressively throughout the system. While the emulator is fast, we do not claim it is the fastest in the world. There are several optimizations left that we have not yet applied. For instance, StarDBT is reportedly faster [22]. However, by design our emulator is very amenable to arbitrary dynamic instrumentation in general and taint analysis in particular. The design of the emulator is our first contribution.

The second key observation is that current DTA approaches are expensive mainly because they need many additional instructions to propagate taint. For instance, every `mov` and `add` incurs substantial overhead. *Minemu* reduces the number of these additional instructions at all cost—sacrificing memory for speed, if need be. Thus, by carefully designing the memory layout, *Minemu* propagates taint at a cost of 1-3 additional instructions. The novel memory layout is our second contribution.

A third key observation is that many additional instructions are due to register pressure in general and tracking taint in registers in particular. Thus, we use SSE registers to track the taint for the processor's general purpose registers—greatly speeding up the taint analysis. Our use of SSE registers is a third contribution.

Because of *Minemu*'s design, the overhead of the taint tracker relative to the emulator is considerably lower than that of other systems, even though we did not yet apply any analysis to prune the taint propagation. Because of this, *Minemu*'s overall performance is also better than that of existing systems, despite the fact that some have faster emulators [22].

Design issues aside, the concrete outcomes contributed by this paper are a very fast DTA emulator based on these insights. The emulator provides a sandbox from which an application cannot escape and offers taint tracking at the byte level. We evaluated the design elaborately with a host of real-world and complex applications (`Apache`, `lighttpd`, connections, `PHP`, `PostgreSQL`, etc.), as well as SPECint 2006 benchmarks. For all real applications, the slowdown was always less than 3x. Often less than 2x. Only one of the SPECint 2006 benchmarks incurred a slowdown greater than 4x, while the overall slowdown across the entire benchmark suite was 2.4x.

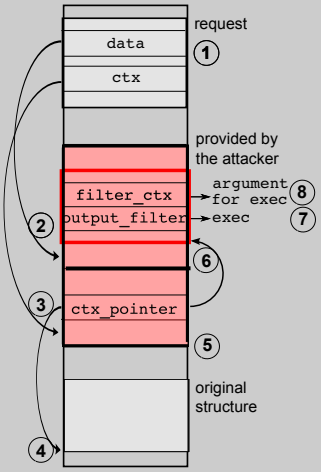
Minemu is real *Minemu* for Linux is available from `https://www.minemu.org`. Interested users can install it today to protect mission critical applications (like `Apache`, `PostgreSQL`, or `lighttpd`) as well as an endless chain of other UNIX tools and shells. To demonstrate the practicality of our emulator, the *Minemu* site (`lighttpd`, `php`, and `PostgreSQL`) itself also runs on the *Minemu* emulator. Moreover, it provides access to a vulnerable `ProFTPD` server, running on *Minemu*, that we encourage readers to attack.

In the remainder of this paper, we discuss the design and implementation of *Minemu* for Linux on 32-bit `x86`. As *Minemu* does not rely on Linux-specific properties, except the size of the address space, porting the design to Windows should be straightforward. We also discuss how the design applies to 64-bit systems.

A buffer underrun vulnerability in `Nginx`

`Nginx` is a web server—in terms of market share across the million busiest sites, the third largest Web server in the world. At the time of writing, it hosts about 23 million domains worldwide. Versions prior to 0.6.38 had a particularly nasty vulnerability.

When `Nginx` receives an HTTP request, it calls `ngx_http_parse_complex_uri` with an `ngx_http_request_t` structure ①. `data` points to a buffer, in which the current routine will store a normalized uri path ②, while `ctx` points to an array of pointers to various context structures ③ and ④. These two buffers happen to be adjacent in memory. The parsing function copies the uri path to `data`, normalizing it at the same time. When provided with a carefully crafted path, nginx wrongly computes its beginning, setting `data` to a location *below* the start of the uri query—somewhere in the buffer underneath it. Next, the user provided query is copied to the location pointed to by `data` ⑤.



Thus, a pointer to a context structure `ngx_output_chain_ctx_t` (`ctx_pointer`) is overwritten with a value coming from the network ⑥. This structure contains a pointer to a function (`output_filter`), which will eventually be called by `Nginx`. By overwriting `ctx_pointer` with a value that points to an attacker controlled buffer, an attacker controls the function pointer, enabling him to load it with the address of the `exec` function in libc ⑦. An adjacent field contains a pointer to this function argument (`filter_ctx`), again controlled by the attacker ⑧. When the function is called, a new program will be executed - picked by the attacker.

Observe that in the above example no code executes in the data segment, so DEP/W⊕X will not help. Moreover, the attack corrupts no canary value, and as the text segment is typically not randomized, ASLR does not stop the attack either.

Fig. 1. A vulnerability in Nginx: DEP, ASLR, and canaries do not stop the attack

2 A new emulator design for fast taint tracking

Minemu is a lightweight process-level emulator designed with taint analysis in mind for the `x86` architecture to protect vulnerable Linux applications efficiently, without special privileges or kernel extensions. *Minemu* runs standard `x86` instructions, so that the application can be written in any language, including assembly.

Attack detection in *Minemu* works just like in other DTA approaches, and taint propagation occurs directly on `x86` instructions. *Minemu* propagates taint as it is copied through, or used as source operand in ALU operations. In addition, it instruments the `call`, `ret` and `jmp` instructions to raise an alert when a tainted value is loaded in `EIP`. Check [20] for the details of the taint propagation rules. This mechanism lets us detect a broad range of all memory corruption attacks. To deal also with code-injection attacks,

which do not need to overwrite critical values with network data, we have extended *Minemu* to check that the memory location loaded on `EIP` is not tainted.

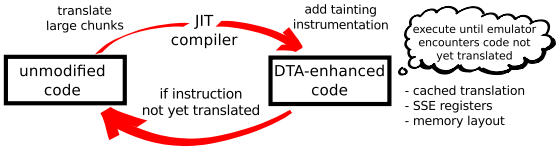


Fig. 2. *Minemu*—high-level overview.

Figure 2 illustrates the big picture. We see that at a high-level of abstraction, *Minemu* is just like other dynamic translators in that it employs a JIT compiler and caches to emulate the underlying processor efficiently. Since the emulated processor is an `x86` itself, *Minemu* will execute as much of the code as possible natively. Whenever *Minemu* encounters an instruction that it has not yet translated, it fetches a large chunk of code to translate it in one go. It resolves all simple branches with targets in the chunk itself, while ensuring that for complicated cases (such as indirect branches), control returns to the JIT compiler. Initially, *Minemu* has not yet translated any instruction, so the first thing it does is translate a maximum sized chunk of instructions—translating until it either reaches the end of the memory area, or encounters an illegal instruction. The size of the translation block is much greater than that of other well-known emulators like QEMU. The translation process also augments the original code with DTA. By caching aggressively, *Minemu* minimizes the overhead of recompilation. Moreover, by using SSE registers instead of the normal general purpose registers for tainting, it alleviates the register pressure that might otherwise occur due to DTA. Finally, the memory layout is especially crafted to make it cheap to propagate taint to the taint map. We discuss all of these aspects in detail in the remaining sections.

Besides dynamic taint analysis (DTA), effective protection against exploits requires the emulator to provide sandboxing of data and code. Specifically, it must confine memory accesses of the emulated process to a designated memory region, to protect *Minemu*'s sensitive data (e.g., the internal data structures and taint values). Similarly, we cannot let the emulated process escape the controlled environment.

In this Section, we discuss the overall design of the *Minemu* emulator, and we continue with the dynamic taint analysis part in Section 3.

2.1 Memory layout

To provide an effective sandbox and implement taint propagation in an efficient way, *Minemu* reorganizes the emulated process' address space.

Figure 3a shows that *Minemu* divides a process' memory into a number of sections. First, an emulated process can only use memory within one contiguous block which starts at the lowest mappable address (*user memory*). It has a size of almost a third of the whole address space. Further, since *Minemu* keeps a one byte taint tag for every byte

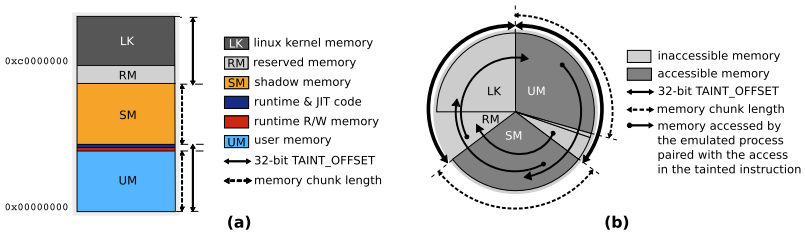


Fig. 3. The figure on the left shows the different sections that make up the address space of an emulated process, while the figure on the right represents the same address space as a circular buffer. As all pointed arcs inside the grey disc have the same angle, they represent a constant offset. So the offset from the start of UM to the start of SM is equal to the offset from the start of RM to the start of UM, etc. We call this distance `TAINT_OFFSET`. Emulated processes can access the dark grey chunks, but an access to a light grey chunk causes a protection error. Whenever a process writes to an address p, *Minemu* adds an instruction to update the taint value in p+`TAINT_OFFSET`—making taint propagation cheap. Suppose a malicious process tries to clean the taint at address p+`TAINT_OFFSET`. Again, during the translation *Minemu* adds an instruction to update the taint value at (p+`TAINT_OFFSET`)+`TAINT_OFFSET`. However, this address is in a protected area (LK) and any attempt to access it leads to a protection error. All sensitive areas are protected in this way—if the process tries to access an illegal memory location, either the operation itself or its corresponding taint propagation instruction causes a page fault.

of the emulated process memory, it reserves a chunk of the same size for the *shadow memory* to store the taint map. In between these chunks, we reserve some memory for the translated JIT code and *Minemu* itself (*runtime & JIT code*), and finally some runtime read/writable data (*runtime R/W memory*). We call the distance between the beginnings of the user and the shadow memory chunks `TAINT_OFFSET`.

Minemu leaves the two final chunks of the address space (*reserved* and *Linux kernel memory*) unused. All memory accesses in these regions generate a protection fault. The combined size of LK and RM is exactly `TAINT_OFFSET`. We will show that reserving this memory and mapping it unreadable allows to run without any boundary checks during emulation. Also, since Linux on the i386 already uses a quarter of the address space for itself, we only reserve/waste a small amount of memory (the *RM* chunk).

While TaintTrace [4] also uses a constant offset for the shadow memory, our layout additonally makes it possible to run *Minemu* without boundary checks during emulation, and still confine memory accesses by an emulated process to user memory (UM).

2.2 Data sandboxing

The memory layout gives each address in user memory a matching one in shadow memory and the distance between them is equal to `TAINT_OFFSET`. During the translation, for each memory access by an emulated process, *Minemu* adds exactly one corresponding memory access which propagates taint to and from the shadow memory. Thus, taint propagation is extremely cheap, as it mainly consists of an instruction accessing memory at a constant offset relative to the original memory location. For example, just before an access to `($eax)`, it inserts an instruction to propagate taint, accessing

(`$eax+disp32(TAINT_OFFSET)`). Similarly, it couples a `push` instruction with an access to (`$esp+disp32(TAINT_OFFSET-4)`).

For data sandboxing, we must confine memory accesses by an emulated process to user memory (UM). Figure 3b shows that when a regular instruction accesses UM, its corresponding taint propagation instruction automatically accesses the corresponding location in shadow memory. Indeed, both operations access memory in the *accessible* sections. However, if a regular instruction tries to manipulate one of the forbidden chunks (the runtime R/W memory, the runtime & JIT code, or the shadow memory directly), the inserted taint propagation instruction will access one of the protected parts of the address space and generate a protection fault. In Figure 3b, these illegal accesses are illustrated with arrows having at least one of its ends in an *inaccessible* light grey chunk. All illegal memory accesses result in page faults—either because of the instruction itself or because of the corresponding taint propagation operation.

2.3 Code sandboxing

Minemu is an emulator using fully dynamic just-in-time (JIT) compilation. When a guest process tries to execute an instruction, *Minemu* translates the code starting at this instruction to produce an equivalent code fragment enhanced with taint tracking. Finally, *Minemu* jumps to the translated code. After executing, control returns to *Minemu* to either locate the next batch of instructions in the cache, or translate them afresh.

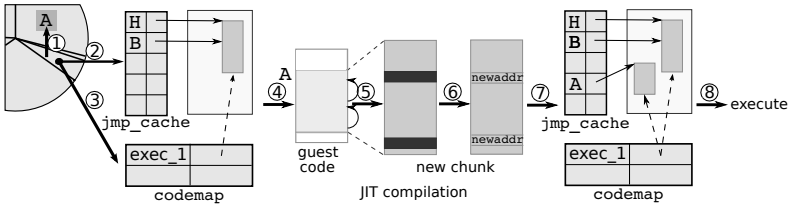


Fig. 4. *Minemu* translation mechanism.

Translation mechanism Figure 4 sketches the code translation procedure. The key steps are cache lookup, used to check whether a guest process code address has been translated before, and JIT compilation, invoked in the case of a cache miss in order to translate a new code chunk. We describe each step below by tracking the way *Minemu* starts executing code that it has not seen before.

In the first step ①, a guest process jumps to a guest code address `A`. *Minemu* searches for a translated chunk corresponding to `A`. It first performs a lookup in the fast cache, `jmp_cache` ②—a hashtable to map jump targets in an emulated process to corresponding addresses in the translated code. Since `A` was not translated before, there is a cache miss, and *Minemu* examines the second table, `codemap` ③. This table contains one row per memory mapped (`mmap`'ed) executable region, and it stores information about translated chunks of a corresponding binary. *Minemu* checks whether `A`

belongs to one of the already translated code chunks. If so, it finds the address corresponding to A, and inserts a new entry in the jmp_cache. In our scenario, however, we assume another cache miss.

Now the JIT compilation process starts ④. Unlike Qemu, fastBT [19] or HDTrans [27], *Minemu* does not translate small blocks of code. Instead, it keeps going until it encounters an illegal instruction or the end of the mmap'ed region. *Minemu* translates from the guest code address A onwards.

When the JIT compiler hits a direct or relative jump instruction, it adds it to a set of *to_be_resolved_jumps*, and continues with the translation ⑤. In Figure 4, the guest code chunk has two jumps, indicated with little arrows. Thus, *to_be_resolved_jumps* contains two elements, depicted as black rectangles in the new chunk.

Once the translation of a chunk of code is complete, *Minemu* examines which jump targets in the *to_be_resolved_jumps* set can be resolved immediately, ⑥. Basically, the JIT compiler determines new jump targets in the translated code for all direct and relative jumps to the same mmap'ed executable region. The rare case of relative jumps across separately mmap'ed sections of a binary is handled separately, but the explanation is beyond the scope of this paper. *Minemu* resolves indirect jumps at runtime. Once hit by an emulated process, they pass the control back to *Minemu*. The emulator handles such jump targets in exactly the same way as the address A in Figure 4. *Minemu* searches the code cache, and provides an appropriate translated chunk to be executed.

When JIT compilation is finished, *Minemu* inserts the newly translated code chunk to both jmp_cache and codemap ⑦. Finally, it starts the execution ⑧.

Additional optimizations To further improve performance, we added a few additional optimizations. The main ones include translated code and return caching.

Translated code caching An optional file-backed caching mechanism can store the translated code. When the executable files of an emulated process are mapped at exactly the same locations as in a previous run of the program, this mechanism allows for reusing code chunks translated earlier. Doing so speeds up programs by eliminating double work. Note however, that we cannot use this optimization in the presence of address space layout randomization.

Return caching The ret instruction is the most common form of an indirect jump. To improve performance, *Minemu* exploits the protocol between the call and ret instructions. Whenever the program executes a call, we can expect a corresponding ret instruction jumping to the program counter following the call instruction. Since the *translated* return address is known at compile time, the JIT compiler simply inserts the right mapping to jmp_cache. If necessary later, *Minemu* is able to retrieve it quickly, without performing a lookup in the codemap cache.

2.4 System calls

Minemu catches all system calls and wraps them to return the control flow to translated code once the execution has completed. Some of them require special handling

by the emulator. For example, when the emulated program invokes `mmap` to allocate new executable memory pages, *Minemu* examines the translated code cache and invalidates entries in this memory region. Specific system calls, e.g., `read` are marked as the sources of taint (e.g., if an emulated process reads from a network socket). It is easy to change the sources of taint in case of different needs for information flow tracking.

2.5 Signal handling

Single instructions from the original program can become multiple instructions in the translated JIT code. This can lead to the kernel delivering a signal while *Minemu* is in a state the original program could never experience. Especially troublesome is the jump cache (`jmp_cache`). If a signal happens in the midst of writing a jump mapping to our cache and the emulated program's signal handler would in the meanwhile look up that address, it could start executing the wrong code.

In order to solve this problem we have implemented a wrapper around signals which allows us to guarantee that signals always get to see a consistent state, as if the program were run natively. The emulator's signal handler uses an alternate stack so as not to disturb any user memory. When a signal comes in, the signal handler checks whether the instruction pointer is between translated instructions that belong to the same original instruction, and whether it is in runtime code.

If the instruction pointer is in the midst of executing an emulated instruction, a JIT translation for that single instruction is made and executed, returning to our signal handler when it is done. In case the instruction pointer is in runtime code or might jump there, we temporarily replace the instruction at which the runtime code jumps back into the JIT code to one that returns to our signal handler.

When the emulator is in a consistent state again, a signal stackframe is copied from the emulator's alternative stack to user memory as if the kernel wrote it there. The original stack frame is then modified to make it reflect the processor state and signal mask as it should be when delivering the user signal so that the following call to sigreturn will actually deliver the signal to the user process' handler.

2.6 Usage

Minemu is a process-based all-user-space emulator. Its invocation is similar to executable wrappers like `nice` and `strace`. Instead of executing the given program, *Minemu* loads it in its own address space and starts emulating it while doing taint tracking at the same time. Child processes and programs started from within *Minemu* will also be emulated the same way. For instance, this is how we start the apache webserver:

```
./minemu -cache /jitcache/ -dump /memdumps/ /etc/init.d/apache start
```

3 Register tagging in *Minemu*

Much of the overhead of earlier DTA systems (e.g., [16, 6, 20]) stems from the large number of additional instructions needed to propagate taint—not just for memory accesses, but also for the registers. Worse still, as the additional instructions require computation to find the location of the taint tags, they typically also increase the pressure

on the x86's already scarce registers. While liveness analysis on registers can mitigate the problem [21], the overhead is still considerable.

By explicitly targeting the x86, *Minemu* is able to exploit architectural features to reduce both the number of additional instructions and the register pressure caused by the instrumentation. Specifically, *Minemu* uses SSE registers to hold the taint information for the general purpose registers to minimize register swapping. As a result, the instructions in need of taint propagation, require as few as $1-3$ extra instructions. In this section, we discuss details of *Minemu*'s register tagging and taint tracking procedure.

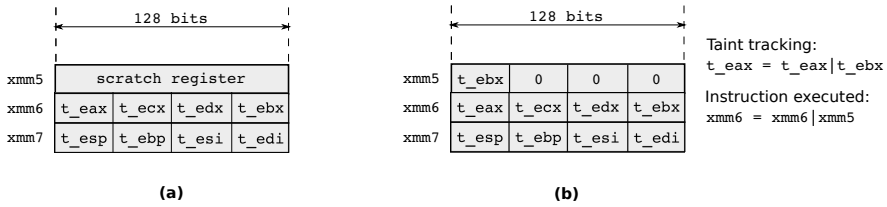


Fig. 5. SSE registers used by *Minemu*. (a) *Minemu* uses three SSE registers to store taint tags of the general purpose x86 registers. t_eax, t_ebx, and so on, denote taint tags associated with the corresponding general purpose registers. (b) An example usage of the scratch register.

3.1 SSE registers used by *Minemu*

To minimize register swapping, *Minemu* emulates a processor without SSE registers, and uses instead three SSE registers to hold the taint information for the general purpose registers. As shown in Figure 5a, two 128 bit registers, xmm6 and xmm7, hold taint values for the eight general purpose registers. Both are conceptually split into four 32-bit parts, and each of these holds the taint value for one of the general purpose registers. We name the taint tags t_eax, t_ecx, and so on. xmm5 is used as an auxiliary buffer, and we call it *the scratch register*. Note that register tagging in *Minemu* is more fine-grained than in most DTA implementations [16, 6, 20]: each individual byte of a register has an associated taint tag, instead of one tag per register.

3.2 Taint tracking

Taint propagation rules in *Minemu* do not differ from those of existing DTA engines. We copy tags on data move operations, or them on ALU operations, and clean tags on common ia32 idioms to zero memory, such as xor $eax,$eax.

What is distinctive about *Minemu* is the way it tracks taint: it does so without swapping out *any* registers. The reason is twofold. First, we use SSE registers to store the general purpose register tags. Second, we do not need to perform any additional computations to determine relevant addresses in the shadow memory. As a result, there is no need to change (and thus to save and restore) the contents of general purpose registers.

As ALU operations are (slightly) more complicated than, say, moves, we will use them as an example. When the emulated process executes `ALU` operations such as `add`, `sub`, `and` or `xor`, *Minemu* inserts instructions to mark the destination operand as tainted if at least one of the source operands is tainted. The tags are thus `or`'ed. Depending on the instruction performed by an emulated process, the destination of a taint propagation `or` instruction inserted by *Minemu* can be either a register or a memory location. For example, an instruction like `$eax:=$eax+$ebx` is coupled with `t_eax|=t_ebx`, and `($eax):=($eax)+$ebx` with `memory_tag($eax)|=t_ebx`, i.e., `($eax+TAINT_OFFSET)|=t_ebx`.

For efficiency reasons, we use the scratch register to temporarily store one of the arguments of the taint propagation operation. Since both cases are handled in a similar fashion, let us assume the destination of the instruction is a register. As depicted in Figure 5b, we first load the taint value associated with the source operand in the scratch register, and place it in the part corresponding to the destination register. The remaining part of the scratch register is zeroed. Now, it suffices to perform an `or` operation on two SSE registers: the scratch register `xmm5`, and either `xmm6` or `xmm7`. By using `xmm5` as an auxiliary buffer, we again manage to avoid swapping out the general purpose registers.

3.3 Is it safe to use SSE registers?

Minemu emulates a processor without SSE registers and instead uses three SSE registers to hold the taint information for the general purpose registers. As not all IA32 processors have SSE registers, compilers and software distributers are often usually very conservative about using them. Even when they are used, there's almost always fallback code for processors that do not support it. If a process *does* try to execute an SSE instruction, *Minemu* currently generates an illegal instruction exception. There is nothing fundamental about this, as it is possible to also translate SSE instructions, by swapping in the contents of the original registers when needed. However, while we have not measured it, it is quite likely that with the swapping overhead, fallback code which does not assume SSE instructions performs better.

4 Evaluation

We evaluate both *Minemu* effectiveness in detecting attacks (Section 4.2) and its performance (Section 4.3). Besides our own measurements, we compare *Minemu* with other fast taint tracking tools (Section 4.4). We also want to mention that *Minemu* is robust. All tested applications worked out of the box.

4.1 Test environment

Our test platform is a quad-core system with an Intel i5-750 CPU clocked at 2.67GHz with 256KB per-CPU cache and 8MB of shared cache. The system holds 4G of DDR3-1333 memory. For our performance tests we used a 32-bit Debian GNU/Linux 6.0 install. Because of library dependencies, some of the older exploits were tested using

Debian GNU/Linux 5.0 or a chrooted Ubuntu 6.06 base install. We tested network applications over the local network loopback device so that our results do not get skewed by bandwidth limitations of the network hardware. We ran each experiment multiple times and present the median. Across all experiments, the 90^{th} percentiles were typically within 10% and never more than 20% off the mean.

In our experiments we mark all input to an application as tainted. Note however, that unlike the other fast tainting approaches ([22, 23, 14]) for *Minemu* the amount of taint does not change the performance at all.

4.2 Effectiveness

Table 1 shows the effectiveness of *Minemu* in detecting a wide range of real-life software vulnerabilities that trigger arbitrary code execution. We mention that, due to the reliability of DTA, *Minemu* did not generate any false positives during any of our experiments. Overall, *Minemu* successfully detects all attacks listed in Table 1. It spots that the program counter is affected by tainted input, and raises an alert preventing the malicious code from executing. Our evaluation shows that *Minemu* detects various types of attacks in real-world scenarios. For example, the vulnerabilities in `Proftpd` and `Cyrus imapd` are exploited to overwrite the return address on the stack and allow remote attackers to execute arbitrary code. For the 2010 `Samba` vulnerability, the attacker uses a buffer overflow to overwrite a destructor callback function. For `Nginx`, an underflow bug on the heap allows attackers to modify a function pointer (as explained in Figure 1). In `Socat` and `Tipxd` it is possible to control the `fmt` parameter to a call to `sprintf`, enabling the attacker to write to arbitrary locations in memory—in this case the return address of a function call.

Application	Vector	Vulnerability	Security adv.	Application	Vector	Vulnerability	Security adv.
Snort 2.4.0	Remote	Stack overflow	CVE-2005-3252	Aspell 0.50.5	Local	Stack overflow	CVE-2004-0548
Cyrus imapd 2.3.2	Remote	Stack overflow	CVE-2006-2502	Htget 0.93	Local	Stack overflow	CVE-2004-0852
Samba 3.0.22	Remote	Heap overflow	CVE-2007-2446	Socat 1.4	Local	Format string	CVE-2004-1484
Nginx 0.6.32	Remote	Buffer underrun	CVE-2009-2629	Aeon 0.2a	Local	Stack overflow	CVE-2005-1019
Proftpd 1.3.3a	Remote	Stack overflow	CVE-2010-4221	Exim 4.41	Local	Stack overflow	EDB-ID#796
Samba 3.2.5	Remote	Heap overflow	CVE-2010-2063	Htget 0.93	Local	Stack overflow	
Ncompress 4.2.4	Local	Stack overflow	CVE-2001-1413	Tipxd 1.1.1	Local	Format string	OSVDB-ID#12346
Iwconfig V.26	Local	Stack overflow	CVE-2003-0947				

Table 1. Tested control flow diversion vulnerabilities

4.3 *Minemu* performance

We evaluate the performance of *Minemu* with a variety of applications—all of the SPECint 2006 benchmarks, and a wide range of real world programs. The slowdown incurred for the SPECint 2006 benchmark is on average 2.4x. The suite of tested real-world applications, in addition to single programs such as `gzip` and `lighttpd`, contains an entire web stack serving over HTTPS. We show that due to the novel emulator architecture, the slowdown incurred for these real-world scenarios is always less than

2.8x, with 1.6x for `gzip`, and less than 1.5x for HTTP/`lighttpd`. In our opinion, the results demonstrate the practicality of our emulator.

Figure 4.3 presents detailed results of our evaluation. The y-axes of all graphs show how many times slower a test was, compared with the same test run natively. In order to measure the overhead of *Minemu*'s binary translator, all of our measurements were done both with and without taint tracking.

In addition to testing single applications, such as `gzip`, `lighttpd`, and `Apache`, we also tested an entire web stack serving over HTTPS. For this test, we chose a PHP-based MediaWiki install running on `lighttpd` and `PostgreSQL`. For `Apache`, `lighttpd` and the MediaWiki web stack we used `apachebench`, and we pinned `apachebench` to a different core than the webserver. For the web stack we also gave `PostgreSQL` a separate core. Doing so decreases request times for both emulated and native runs and reflects what real installations would do.

We observe that the slowdown incurred by `lighttpd` serving HTTP is minimal, always less than 1.5x, and decreasing with the size of a request. This illustrates that for IO-bound applications, like serving documents over HTTP, the cost of taint tracking using *Minemu* is minimal. In the case of HTTPS, the slowdown increases with the size of a request, but is still less than 2.8x for large files.

We also ran the whole SPECint 2006 to see the effect of *Minemu* on applications which do not spend a lot of time waiting for input. Because the SPECint 2006 benchmarks are CPU intensive, and spend most of their time doing hard computations, we expect these results to represent worst case scenarios. Nevertheless, only one of the SPECint 2006 benchmarks, `h264ref` - performing video compression, incurred a slowdown greater than 4x. Moreover, eight out of twelve benchmarks incur a slowdown ranging from 1.7x to 2.3x.

4.4 How does *Minemu* compare to related work?

In this section, we compare the performance of *Minemu* with three systems that are the most relevant to our work, PTT [14], the dynamic taint tracking tool by Saxena et al. [23], and LIFT [22]. We refrain from discussing the details of these projects until Section 6 and focus on performance only. We will see that *Minemu* outperforms all. In all graphs in this section, *Minemu*-T, and *Minemu*-NT denote the results of *Minemu* with- and without taint tracking, respectively.

PTT PTT [14] is a taint tracking system which, similarly to [15], dynamically switches execution between a heavily instrumented QEMU and fast Xen, depending on whether tracking is required. As we shall see, even though PTT has numerous optimizations to reduce the performance overhead, *Minemu* is much faster.

To evaluate the performance of PTT, its authors present three benchmarks: local copy, compression and searching. Local copy involves copying of a 4 MB file using the `cp` command, and compression - compressing a 4 MB file with `gzip`. As for searching, the `grep` command is used to search the input data for a single word. The input data set is a 100 MB text corpus spread across 100 equal-sized files. Figure 7 compares the slowdowns incurred by PTT, and *Minemu*. Since the `cp a-4MB-file` operation

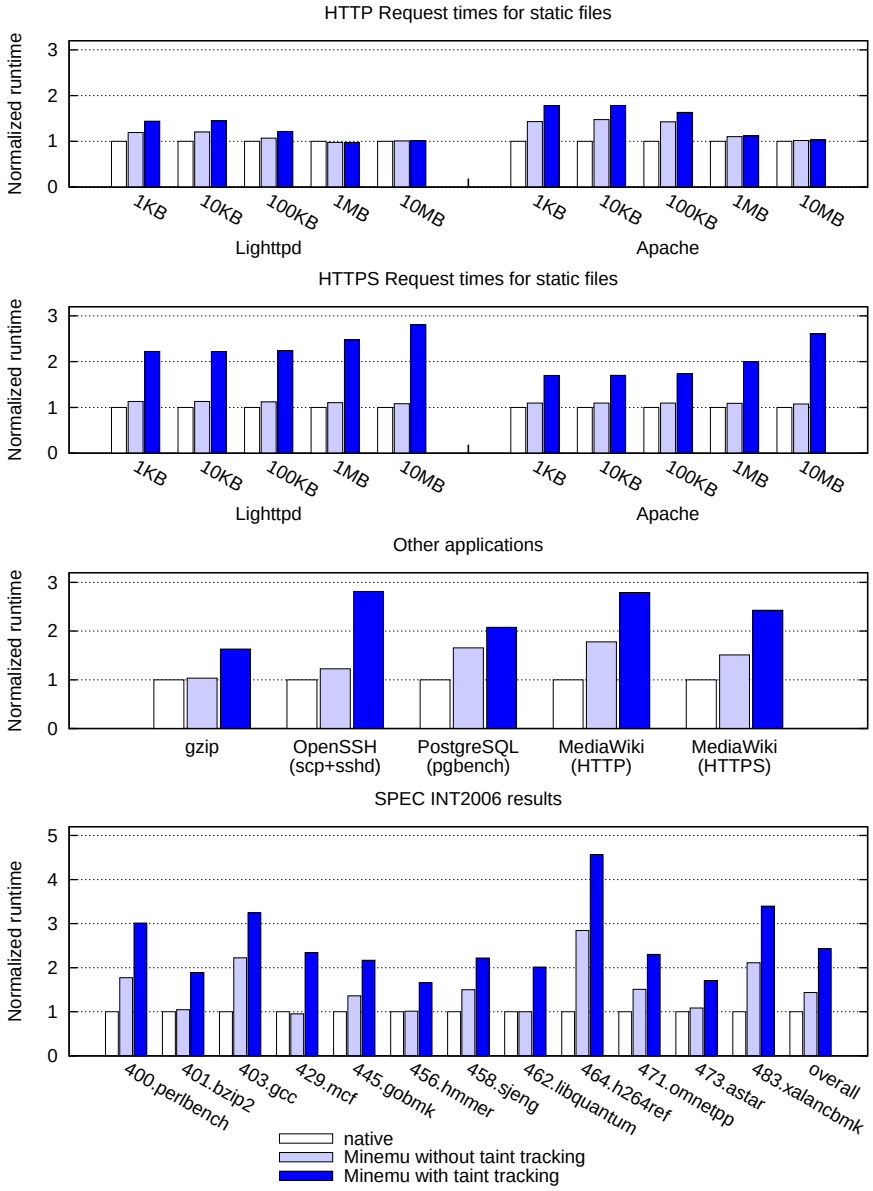


Fig. 6. Overhead of emulation and taint-tracking in *Minemu*, compared to the native execution.

is dominated by the initialization time, we also present *Minemu* overhead for a `cp a-100MB-file` operation. We can see that in all cases, *Minemu* significantly outperforms PTT. Note, however, that PTT does full system emulation rather than process emulation.

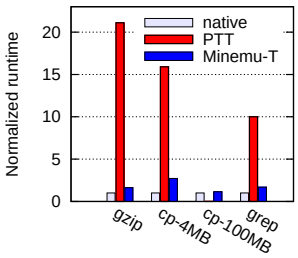


Fig. 7. Comparison of performance overhead incurred by PTT and *Minemu*.

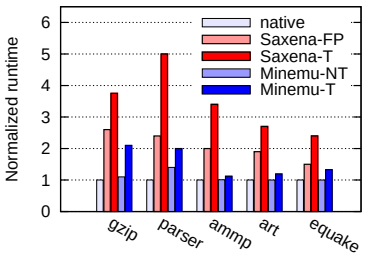


Fig. 8. Comparison of performance overhead incurred by Saxena et al. [23] and *Minemu*. `gzip` and `parser` come from SPECint 2000, `ammp`, `art` and `equake` from SPECfp 2000.

Saxena et al. The fast taint tracking system by Saxena et al. [23] builds on smart static analysis. This may be a problem, because as we discuss further in Section 6, the information required by the static analysis is not always available in practice.

To evaluate the performance of the system, the authors run a rather eclectic mix of ten SPEC benchmarks. As some of them are so old as to be hard to find (SPEC 92 and SPEC 95), we were not able to fully compare *Minemu* with [23]. Four of the applications evaluated in [23] are SPECfp benchmarks. Since FPU registers are rarely, if ever, involved in attacks, most taint tracking systems, including *Minemu* and Saxena et al. [23], ignore them by default. Thus, the overhead stems only from the *usual* taint tracking instructions, such as data movement, arithmetic or logic instructions. For the sake of comparison only, we present *Minemu* results for these applications as well.

Figure 8 compares slowdowns for the benchmarks which we had available. The results show the overhead of [23] in two cases, first, optimized taint-tracking (Saxena-T), and second, *fastpath* (Saxena-FP). Similar to LIFT, [23] also optionally implements *fastpath*. Before executing a basic block it checks whether the data involved is tainted or not. If not, execution follows a fast binary version without any information flow tracking. The authors of the system measured the performance of the fastpath and slowpath code separately, where the fastpath results do not involve tainted data tracking. Whenever we do have means for comparison, *Minemu* is significantly faster. Even with full taint tracking, *Minemu* performs better than the Fastpath version of [23].

LIFT LIFT [22] implements taint analysis in Intel's highly optimized StarDBT binary translator and applies three taint tracking performance optimizations. We show that

although currently *Minemu* does not apply any of these optimizations, in most cases it performs better. We also point out that the overhead added by the taint tracking relative to the performance of the bare emulator is significantly lower in the case of *Minemu*.

To evaluate the performance of LIFT, its authors measured the throughput and response time of the `Apache` web server, and run 7 (out of 12) SPECint 2000 benchmarks. Refer to Figure 9 for slowdown comparisons. The overall overhead incurred by *Minemu* is much lower than that of LIFT with `gcc` as the only exception. *Minemu*'s performance when running `gcc` ranges from 2x to 3.9x (*Minemu* compiles itself in about 2x native on our Intel i5-750 CPU), and differs from system to system for the same program. Since the performance is also poor for *Minemu* without taint analysis, it is not likely to be caused by the working set not fitting into cache memory. Rather, it is probably an emulator problem. Other emulators, such as StarDBT, perform better on this benchmark. It shows that there is room for improvement in our emulator implementation. We also observe that even though StarDBT is mostly faster than our pure emulator, the taint tracking mechanism implemented in *Minemu* incurs less additional overhead.

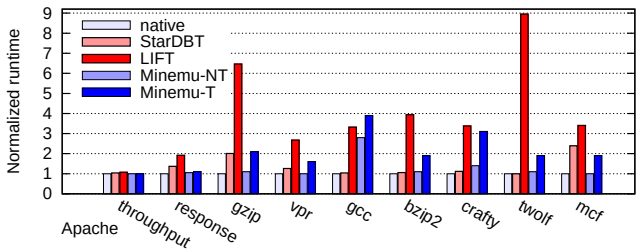


Fig. 9. Comparison of performance overhead incurred by LIFT [22] and *Minemu*.

5 Limitations and Future work

Limitations *Minemu* suffers from the same drawbacks as most other DTA implementations: it does not track implicit flows, and it does not detect non-control data attacks. In addition, *Minemu* consumes more memory than existing approaches. Extremely memory-hungry applications may not be very suitable for *Minemu* in its current form. In the next section, we discuss how the *Minemu* architecture applies to 64-bit architectures with larger address spaces.

Also, *Minemu* currently does not support self-modifying code. A possible solution is to use the write protection mechanism. Executable pages are marked unwritable, so that whenever an emulated process modifies the original code, *Minemu* would take control of the execution. By removing all entries which correspond to the modified user code page from the translated code cache, the new code will be translated by the JIT compiler before the emulated process executes it. We leave it as future work.

Finally, the current implementation does not work for applications that insist on using SSE instructions. However, we do not consider this a fundamental problem, as it is straightforward to implement register swapping for these cases.

Minemu for a 64-bit architecture Although our approach is particularly well suited for 32-bit `x86` code, we believe we can make it work efficiently on 64-bit `x86` also. The main obstacle is that while on a 32-bit system we can easily pretend that our emulated CPU does not support SSE extensions, they come standard on 64-bit `x86`. As a result, any compiler is free to make use of them without any feature checking. Fortunately, the latest Intel and AMD processors come with even wider vector registers suitable to hold taint data[3]. However, because the lower 128 bits of these registers map to the old SSE registers, we will need some swapping for lesser-used registers.

A second problem is that the 32-bit displacement in Intel's addressing mode used for `TAINT_OFFSET` is not large enough to hold the whole address space. This is no problem as long as a program does not try to allocate consecutive regions of memory of more than 2G in size. By interleaving normal memory and shadow memory in chunks of 2G we can still use the same mechanism for tainting. If we want to support more than 2G of consecutive memory, the emulator should reserve one (less-used) general purpose register to hold `TAINT_OFFSET`. Memory accesses which do not use base-index addressing can be translated into a base-index address with `TAINT_OFFSET` as base. Accesses which do use base-index addressing will need an additional `lea` instruction.

6 Related work

Binary instrumentation for taint tracking Dynamic taint analysis builds on seminal work by Peter and Dorothy Denning on information flow tracking in the 70s [10]. Since then we have witnessed a string of publications discussing taint tracking, e.g., TaintCheck [16], Vigilante [6], XenTaint [15], and Argos [20]. As all these systems, however, are too slow to be used in production systems, researchers started working on optimizations that would render dynamic taint analysis useful in real world scenarios. In this section, we discuss three recent approaches which aim at decreasing the overhead incurred by DTA: the work by Saxena et al. [23], LIFT [22], and PTT [14]. We compared the performance of *Minemu* with these systems in Section 4.4, and we showed that *Minemu* outperforms all of them. We focus on the architecture of these tools now.

State-of-the-art performance optimization for taint analysis by Saxena et al. [23] builds on smart static analysis. Prior to execution, it translates the original binary to a completely new binary that adds highly optimized instrumentation code only to instructions that really need it. Unfortunately, even static disassembly of stripped binaries is still an unsolved problem. For this reason, the analysis assumes the presence of at least some symbolic information (like the entry points of functions), which is typically not available in practice.

LIFT [22] implements taint analysis in Intel's highly optimized StarDBT binary translator. StarDBT uses additional dedicated registers for taint tracking. Specifically, it

[3] `http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/intref_cls/common/intref_avx_details.htm`

translates the IA32 instructions to EM64T binary code. Since the EM64T architecture has more registers than the IA32, StarDBT does not need to spill registers, giving a significant performance gain. As a consequence, however, LIFT will not work on a 32-bit installation. LIFT applies three additional performance optimizations. First, before executing a basic block LIFT checks whether the data involved is tainted or not. If not, execution follows a fast binary version without any information flow tracking. Second, LIFT coalesces data safety checks from multiple consecutive basic blocks into one. Third, LIFT reduces the overhead of switching between the emulated program and the instrumentation code by using cheaper instructions and status register liveness analysis, respectively. While *Minemu* does not apply any of these optimizations (yet), in most cases it performs better already. If anything, they show that *Minemu*'s performance can be improved even more. Also, our overhead for (just) the taint tracking is lower.

PTT [14] is a taint tracking system which, similarly to [15], dynamically switches execution between a heavily instrumented QEMU and fast Xen, depending on whether tracking is required. To improve performance, PTT tracks taint tags at a higher abstraction level and in an asynchronous manner. In some more details, instead of instrumenting the micro instructions generated by QEMU, PTT creates a separate stream of tag tracking instructions from the `x86` instruction stream itself. Since the emulation and taint tracking are now largely separable, PTT executes the taint tracking stream in a parallel asynchronous fashion. This results in a significant performance gain. Still, *Minemu* greatly outperforms PTT.

Binary translation Binary translation has been an important research topic for at least 30 years [11] now. In this section, we limit ourselves to two systems which are most similar to *Minemu*, fastBT [19] and HDTrans [27]. Both systems are light-weight process emulators that use code caches for translated code, and apply efficient optimizations for indirect jumps. Since *Minemu* is more than an emulator - it employs binary translation to provide efficient taint tracking - we cannot perform a comprehensive comparison with the aforementioned emulators. We focus the discussion on the main design decisions. Whenever relevant, we also refer to QEMU [2]. Even though QEMU uses binary translation to implement full system virtualization, it has been used as a basis for multiple taint tracking tools, e.g., Argos [20].

Compared to these three system, *Minemu* translates the longest chunks of code at a time. It stops only at the end of a memory region or at an illegal instruction. In principle QEMU and fastBT translate basic blocks, while HDTrans stops at conditional jumps or return instructions. Another important aspect of binary translation tools is the way they handle indirect jumps, and the issue of return caching. *Minemu*'s handling of indirect jumps is most similar to HDTrans - both systems use a lookup table that maps locations in the code cache to locations in the original program. Keep in mind however, that in *Minemu* translated code chunks are much longer than in HDTrans, so that many jump targets are located inside chunks. As for the return caching mechanism, all three emulators implement mechanisms that exploit the relationship between `call` and `ret` instructions to efficiently cache the return address.

7 Conclusions

In this paper, we explored the research question of whether or not the slowness of software dynamic taint analysis is fundamental. We believe that we have (at least partially) answered this question in the negative. An emulator that is carefully designed explicitly for taint analysis, achieves significant speed-ups. We developed *Minemu*, a fast taint-tracking `x86` emulator and showed that the slow-down caused by the combination of taint analysis and emulation ranges between 1.5x and 3x for real applications. The design introduces a novel memory layout that minimizes the overhead for propagating taint in memory operations. In addition, it uses SSE registers to alleviate potential register pressure due to the instrumentation. We evaluated our solution with standard benchmarks as well as suites of real and complex software stacks. Finally, we compared our results with other approaches towards speeding up DTA and show that *Minemu* is significantly faster. *Minemu* is available for download from `https://www.minemu.org`. Because of its excellent performance, we believe that *Minemu* may make DTA suitable for production machines.

Acknowledgments

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, as well as by the European Commission through projects FP7-ICT-257007 SYSSEC and iCode (funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security- related Risks Programme of the European Commission Directorate-General for Home Affairs). This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein. We are grateful to David Brumley and his team for several of the local exploits we used to evaluate *Minemu*. We would like to thank Georgios Portokalidis for fruitful discussions, and the anonymous reviewers for useful comments.

References

1. CVE-2009-2629: Buffer underflow vulnerability in nginx. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629, 2009.
2. F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference*, 2005.
3. S. Bhatkar, D. D. Varney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th USENIX Security Symposium*, pages 105–120, August 2003.
4. W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the 11th Symposium on Computers and Communications*, 2006.
5. J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conference*, 2008. Best Paper Award.
6. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *Proc. of SOSP'05*, 2005.

7. C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, 1998.
8. J. Crandall and F. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th Interational Symposium on Microarchitecture*, 2004.
9. M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, 2007.
10. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
11. L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proc. of the 11th Symposium on Principles of programming languages (POPL)*, 1984.
12. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *ATC'07: 2007 USENIX Annual Technical Conference*, 2007.
13. W. Enck, P. Gilbert, B.-G. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smart phones. In *Proceedings of OSDI'10*, Vancouver, BC, October 2010.
14. A. Ermolinskiy, S. Katti, S. Shenker, L. L. Fowler, and M. McCauley. Towards practical taint tracking. Technical Report UCB/EECS-2010-92, University of California, 2010.
15. A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proc. ACM SIGOPS EUROSYS'2006*, 2006.
16. J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proc. of NDSS*, 2005.
17. A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
18. PaX. Pax. `http://pax.grsecurity.net/`, 2000.
19. M. Payer and T. R. Gross. Generating low-overhead dynamic binary translators. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, 2010.
20. G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, 2006.
21. M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *Proc. of WCRE'02*, 2002.
22. F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of MICRO*, 2006.
23. P. Saxena, R. Sekar, and V. Parunik. Efficient fine-grained instrumentation with applications to tain-tracking. In *In Proc. of ACM CGO'08*, Boston, MA, April 2008.
24. E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, SP'10, 2010.
25. Secunia. DEP/ASLR implementation progress in popular third-party windows applications. `http://secunia.com/gfx/pdf/DEPASLR2010paper.pdf`, June 2010.
26. A. Slowinska and H. Bos. The Age of Data: Pinpointing guilty bytes in polymorphic buffer overflows on heap or stack. In *Proc. of ACSAC'07*, 2007.
27. S. Sridhar, J. S. Shapiro, and E. Northup. Hdtrans: An open source, low-level dynamic instrumentation system. In *Proc. of VEE'06*, 2006.
28. G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI*. ACM, 2004.
29. W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, 2006.
30. H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS '07*, 2007.