The Page-Fault Weird Machine: Lessons in Instruction-less Computation

Julian Bangert, Sergey Bratus, Rebecca Shapiro, Sean W. Smith

Abstract

Trust Analysis, i.e. determining that a system will not execute some class of computations, typically assumes that all computation is captured by an instruction trace. We show that powerful computation on x86 processors is possible without executing *any* CPU instructions. We demonstrate a Turing-complete execution environment driven solely by the IA32 architecture's interrupt handling and memory translation tables, in which the processor is trapped in a series of page faults and double faults, without ever successfully dispatching any instructions. The "hard-wired" logic of handling these faults is used to perform arithmetic and logic primitives, as well as memory reads and writes. This mechanism can also perform branches and loops if the memory is set up and mapped just right. We discuss the lessons of this execution model for future trustworthy architectures.

1 Introduction

Computing architectures are typically described in terms of their instruction set architecture (ISA). Formal security models of processors generally focus on the semantics of CPU instructions in an ISA (e.g., Morrisett's RockSalt [13]). Thus it is easy to see a sequence of machine instructions as the only vehicle of computation in a processor; it is easy to expect successful dispatching of instructions to be a necessary condition for having a non-trivial computation. After all, if no instructions have been successfully executed, what work could a processor have possibly done?

However, modern microprocessors have many other mechanisms that are able to perform a surprising amount of computation. We demonstrate that the page-fault handling mechanism in the Intel's IA32 architecture—in combination with a few other legacy features—is able to perform Turing-complete computation without *any* CPU instructions completing. We believe that this is not unique to either the x86 Memory Management Unit (MMU) or to the Intel architecture, but that similarly complex architectures have similarly interesting behavior outside their "main" instruction set.

Although our proof-of-concept represents neither a vulnerability in IA32, nor an exploit for x86 processors, but we believe it continues the line of research that originated in exploit development—namely, exposing unexpected (and unexpectedly powerful) programming models within the targeted environments, where programming happens via maliciously crafted data rather than with native binary code. We show that CPUs carry within them a "weird machine" programming model which does not rely on any actual CPU instructions.

We believe that understanding such unexpected "weird machine" execution models is necessary in order to work toward establishing the trustworthiness of a system. Indeed, trust in a system can be characterized as assurance that certain kinds of potential computations are not actually possible. Analyses of trust typically assumes implicitly that the universe of potential computation consists only of things expressible as execution or access traces. By showing the existence of computations outside this universe, weird machines violate these assumptions and serve as proofs of non-trustworthiness.

Lessons and impact. We initially undertook this study to better understand x86 trapping, which is the foundation of core OS security mechanisms. Indeed, both verification of software and the formal study of enforcible policies rely on certain assumptions regarding the underlying memory model; it's the trapping of memory accesses that actually enforces such assumptions. However, x86 memory trapping is driven by many inputs, has substantial amounts of state that is affected by those inputs, and also *writes* memory—thus leading to the presence of non-trivial memory-modifying automata in x86 memory trapping. What better way to understand the system than try to program automata to extract maximum

possible computing power from it?

Our "programs" exists as a set of cross-referencing and cross-mapped memory tables; all entries of these tables are well-formed, and use no undocumented processor features. The existence of these programs poses a number of questions:

1. What makes these tables as a whole well-formed or benign?
2. If machine owners want to exclude the kinds of computation we describe, how should they go about it?
3. Are there formal security models that implicitly or explicitly assume that the memory management hardware cannot be driven through a Turing-complete computation on its input tables?
4. Should future MMU deliberately limit computational power by design?
5. Since trap descriptor tables may enable complex programming models, should they be treated the same way as covert channels?
6. How can adversaries use the existence of this weird machine to cause actual damage?

We believe that the designers of a trustworthy platform should start considering these questions so they could choose to incorporate the answers as a part of their security model.

We hope that the obvious obfuscation and computation hiding potential of our construct will lead to both interesting obfuscation techniques and inform the design of new security features.

2 Related work: exploits and hidden processor state

2.1 Inspirations from exploit programming

Red pills. In the process of developing our prototype, we encountered many bugs in emulation and tracing tools (such as QEMU); in a sense, our Turing machine is *made* out of "red pills" capable of detecting the type of environment it is running in. Not a single such emulation and tracing tool reflected the entirety of hardware's actual behavior on which we rely. We believe the reason for this limitation is that emulation-based analysis tools are primarily debugged against popular OS implementations rather than hardware specifications. Although "red pills" are plentiful and can be automatically produced (see, e.g., [16]), the ability to host arbitrary computations in the "red pill" space is a fresh cautionary tale.

Weird machines. Leading hacker researchers long held the idea that exploitation was a a form of programming by composing the target platform's features and bugs to carry out unexpected or arbitrary computation (e.g., [21, 10, 5]). The bugs, triggered in ways to minimize and control their effects, yielded instruction-like *primitives* such as reading or writing a word at a specified location. Just as an assembly program combines native instructions, exploit payloads were constructed out of computational primitives exposed by both features and bugs.

Bratus et al. [3, 4] coined the term *weird machines* to refer to such programming models that are the underlying computational reality of exploits, to capture the rich folklore of hacker research that dealt with these modes of programming. Just as programming in native assembly relies not only on the instruction set but also on programming idioms, a weird machine is a programming model built out of the collection of primitives exposed by the target as an "instruction set", and its usage idioms.

Classic low-level examples of such idioms and primitives that add up to a rich programming model include:

- *Format string exploitation* (in which the internals of `printf()` served as the automaton and the crafted format string as a program driving that automaton to corrupt runtime memory in controlled fashion),
- *Heap metadata exploits* [2, 11, 10] (in which heap management code was the simple automaton programmed to perform memory writes by overflowing a heap block's freelist pointers),
- *Crafted stack frames* that serve as a program to the automaton composed of a collection of "gadgets" of Return-Oriented Programming [24, 22, 6] (which, in its early forms [20, 14], were simply parts of the automaton implementing the program's own control flow). All of these examples operate on the level of code compiled into native instructions and loaded into a process' runtime space.

Recent web exploitation techniques expose "weird machines" higher up in the software stack – in web browsers where exploit execution depends entirely on features of the web browser rendering engine (DOM, CCS, HTML5, etc.) and browser components or the server web programming environments, never triggering any native binary-level bugs in the browser binary. These execution models leverage the fact that rendering various elements of the Document Object Model (DOM) tree has rich and observable side-effects on the browser state, while the browser also contains automation logic that can be repeatedly triggered (see, e.g., Heiderich's [9, 8]).

Lower down in the software stack, ABI metadata provide more examples of Turing-complete programming

with crafted inputs that aren't, strictly speaking, malformed. For example, ELF relocation entries in combination with the dynamic symbol tables can be used as a program executing on the ELF runtime linker-loader [19].[1] DWARF exception handling data used by the GCC toolchain for encoding stack unwinding and saved register information can be used to drive an arbitrary computation on the exception-handling standard library code [15]. In both cases, the metadata plays the role of instructions that drive the "weird machine" automaton present in the target (in the code that processes the respective kinds of metadata to create or modify process memory and/or stack).

The variety of examples of weird machines we have discussed show that there are rich – and sometimes Turing-complete – programming models present in all layers of the software stack.

Ubiquity of weird execution environments. From the "weird machine" point of view, *every input is a program*, so long as it causes state changes in the system that consumes it. This view of input is, of course, standard in computation theory: a Turing Machine in of itself merely holds potential computing power until it presented with some input to drive it. Finite automata and pushdown automata that recognize their respective input languages can be thought of as driven from state to state by the input symbols they consume, and so on.

It is not as common, however, in software engineering to view inputs as programs: network packets are not seen as inputs that are executed by the network stack, document formats on their respective processing applications, ABI metadata on the loaders and runtime linkers, RPC messages on their interpreters, encrypted messages on the cryptographic transport libraries, and so on. It appears that idea of treating inputs as languages to be recognized by automata-based parsers generated from the specification grammars is limited to the narrow fields connected with programming languages research (e.g., [28, 12]). Sassaman et al. [23] outlines the implications of this ad-hoc approach to input processing for software security, and identifies it as a major contributing factor to input-related vulnerabilities.

The majority of "weird machines" described have been in software environments, such as application process runtime or the operating system kernel. In the next section, we discuss examples of programming automata inside x86 hardware components that served as inspiration for our weird machine construction.

[1] This work generalized the LOCREATE proof-of-concept [25], which used the automaton underlying PE relocation to encode an "unpacker" (a binary code rewriter) solely in crafted relocation entries.

2.2 A case study of programming MMU state

It is not surprising that memory hierarchy architectures of modern processors maintain additional state to optimize virtual memory address translation. It is remarkable, however, that this state can be reliably controlled by causing specific sequences of memory accesses on crafted page-tables. Since this state affects memory access trapping, these traps can be thought of as having additional semantic features available to the systems programmer.

On x86, these features have been used as either a powerful security primitive [17], a debugger aid [27], or a rootkit memory hiding trick [26]. In this section, we explain these different uses; readers familiar with these results are encouraged to skip to Section 3. Note that we do not rely on any of the following mechanism for our construction, rather they served as an inspiration to the computational power of the Intel MMU.

An x86 "split TLB" primer The X86 memory architecture uses separate caching paths for fetching data and instructions, each with its own state preserved across a history of memory accesses. The page table entry (PTE) of a successful data memory reference is lifted into the data translation lookaside buffer (dTLB) whereas successful instruction fetches resulting from control flow transfers such as `jump` or `call` instructions get cached in a separate instruction translation lookaside buffer (iTLB). Once a TLB entry is created for a virtual memory page in the appropriate TLB, *it will be used for address translation whether or not the underlying PTE is changed*; the PTE record will only be consulted again after the TLB entry is evicted or flushed. Thus the PTE entry for a page may be different from either or both of the TLB entries for that page currently in use; it turns out that this condition can be controlled and the associated logic used as a programming primitive, as the following examples show.

PaX PAGEEXEC. The PAGEEXEC [17] mechanism of the PaX project is a prime example of using the semantics of TLB state as a security primitive. Prior to the broad introduction of the NX bit on x86 platforms, PaX emulated non-executable pages by using x86 segmentation where available or PAGEEXEC where segmentation was not an option. With its demonstrated efficacy against executable-stack exploits, this property gained recognition and guided the deployment of OS security features such as OpenBSD's $W \oplus X$ and Microsoft's DEP.

The PaX team performed a careful analysis of TLB and trapping logic, and demonstrated that it indeed provided the necessary semantics. We refer the reader to the

PAGEXEC documentation [18] for the full formal description of the automaton involved; we briefly describe how it works below.

In the absence of a dedicated non-executable data support, the 'non-executable' property of a page can be emulated, so long as all potential instruction fetches from that page are trapped and examined by the trap handler (if the EIP register value at the point of the trap is inside the page's virtual address range, it's a fetch). However, since invoking a trap handler incurs a heavy performance penalty, the trapping cannot be allowed to occur for each and every access to the page. Luckily, this is where page fault logic and memory address translation semantics compose into an efficient solution.

By setting the Supervisor/User (S/U) bit in the PTE of a designated non-executable page, we can cause the processor to trap any access to that page, i.e., whenever a virtual address within that page goes through the page translation look-up process. The trap causes the Page Fault handler to be invoked (as no TLB has a corresponding entry yet). If the resulting trapped page access is due to an instruction fetch, then the page fault handler terminates the process; otherwise, it's a data access and must be allowed through. The page fault handler then resets the S/U bit for a single data byte access to succeed, and performs that access – causing the PTE for the page to be recorded in the dTLB. Right after this access, the handler resets the PTE entry's S/U bit back to unconditionally causing the fault. Subsequent dTLB-cached accesses will succeed incurring no penalty until the dTLB entry is evicted, whereas all other accesses, including all instruction-fetch accesses, will fault. Upon dTLB entry eviction, the overloaded page fault handler will be called and will restore the dTLB entry "de-synchronized" from the iTLB and the PTE entries.

Thus the "de-synchronized" PTE and dTLB entries together with the Page Fault logic can be used to introduce extra page trapping semantics – in this case, the one that the NX bit later provided natively.

OllyBone. OllyBone [27] used a similar mechanism as the basis of a malware-analysis debugger module to trap "packed" malware right after the unpacker has extracted ("unpacked") the actual malicious code. As its author observed while analyzing malware, a desired trapping primitive would be catching the first instruction executed that did not exist in the packed file – that is, an instruction previously written by the unpacker. This would be possible if these newly created instructions could somehow be tagged, and the MMU could be configured to trap on the tag.

While this sounds like a complex proposition for experimental hardware, OllyBone solved this problem by using the TLB property described above to act as that tag. This created a practical approximation to trapping on the condition "instruction fetch from a page previously written to by the program," a convenient means to recover code obfuscated by complicated packers.

ShadowWalker. The "Shadow Walker" technique leveraged the split TLBs for the purpose of concealing pages of executable code in virtual memory from a process such as an anti-virus that would analyze the code (accessing it as binary data). The gist of the technique was to de-synchronize the physical page frame number (PFN) in iTLB and dTLB in such a way that the CPU's memory reads within the hidden range of virtual addresses would be translated to a different physical page than instructions fetched from the same virtual address range. Thus an anti-virus or a kernel debugger would in fact be "analyzing" the contents of an innocuous page, while `jump`-ing or `call`-ing into that page would execute entirely different code.

In all of these examples, the interaction of the memory trapping mechanisms with the caching layer of memory translation has introduced additional and powerful semantics, enabling programming tricks that would seem impossible from a naive view of either mechanism. Many such composition effects are common folklore in hacker research; we posit that they deserve a formal study, starting with descriptions of the programming models they enable. This, in part, is what motivates our work described in the next section.

3 Implementing Interrupt-Based Computation

3.1 Overview

Memory translation and interrupt handling on the IA32 architecture are controlled through a combination of architectural registers and tables in memory. Furthermore, when a page fault occurs because of an invalid page-table entry, information about the fault is written to a location in the same memory. By configuring the address of the page fault handler to be yet another invalid address, the processor will keep endlessly dispatching page-faults as it tries to fetch the first instruction of the page-fault handler. If the tables controlling this behavior are crafted just the right way, the side effects of the interrupt handling form a Turing-complete one-instruction computer.

This suggests that we could consider view the internal logic of page fault and memory translation as the finite automaton of a Turing machine and the memory as holding its 'tape'", by potentially creating a kind of a closed loop of memory accesses.Figure 1

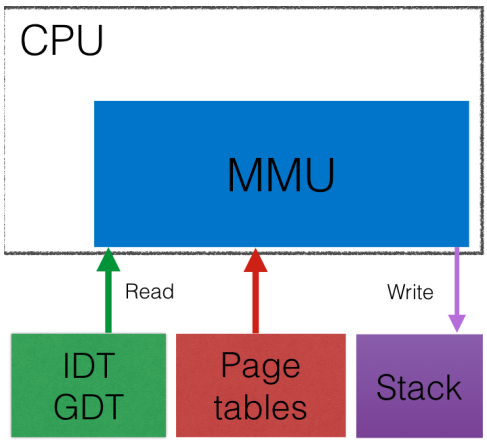

Figure 1: Closing the computation loop.

3.2 X86 Interrupt Handling Refresher

The interrupt handling mechanism of the x86 is closely related to the now-disused protected-mode[2] memory segmentation system[1]. This system is driven primarily through two tables in memory, the Global Descriptor Table (GDT) and the Interrupt Descriptor Table (IDT). The GDT contains segment descriptors, i.e. intervals of memory addresses each labeled with a descriptive type. The CPU addresses memory through an index into the GDT, called a *segment selector* and an offset into the resulting interval. Usually, the segment selector for a particular access is stored in an implicit register, such as how instruction fetches use the selector from the Code Segment (CS) register. In modern processors, this feature has been de-optimized and is largely unused, and will not be used in our construction , so we will disregard it.

More interestingly, the GDT can also point to Task State Segments (TSS) which are regions of memory that can contain a copy of most of the processor's registers in addition to additional control information, such as the location of the kernel stack to be used for handling interrupts (see Figure 2). The CPU has hardware features that allow saving the entire CPU state to a TSS and restoring it from there. Therefore, the TSS mechanism allows the system programmer to switch between contexts with a single instruction or interrupt without any further manual task switching logic. [3].

[2]The techniques presented here are constrained to 32-bit mode. While many systems now run in long mode (64 bit), the 64-capable chips still support all 32 bit features and can be returned to 32-bit mode. Furthermore, we believe that similar computational power also lies in 64-bit mode interrupt handling.

[3]In practice, this is unused due to the performance overhead compared to handwritten task switching logic. Furthermore, additional glue code is needed to properly save and restore additional architectural state added after Protected Mode, such as floating point registers

The Task Register (TR) contains the selector for the 'current' task state segment. Whenever the CPU switches to a different task, before loading its state from the new task's TSS, it will save its state to the TSS stored in TR.

The other important table for protected mode is the Interrupt Descriptor Table, residing at a virtual address stored in a special register. Each IDT entry corresponds to a class of interrupts and contains instructions how to handle these interrupts. One the one hand, the processor can be instructed to leave most state untouched, move to a privileged code segment and kernel stack and jump to an interrupt handler at a given address. On the other hand, certain interrupts might be symptomatic of kernel bugs that have seriously corrupted the CPU state, in which case the IDT can tell the processor to perform a task switch to a specified task. The task switching mechanism reloads more state than the normal FAR-jump interrupt gate, so it can recover more state corruption.

For some interrupts, the processor pushes an *error code* to the interrupt handlers stack. If an error occurs while transferring to the interrupt handler, the CPU raises a double fault interrupt. If yet another interrupt occurs while handling the double fault, the CPU resets itself and the system reboots.

3.3 Constructing the Weird Machine

We can use the primitives listed above to assemble a "One Instruction Computer" with a move-branch-if-zero-or-decrement instruction, short *movdbz*.

Each *movdbz* instruction consists of a source, a destination, a branch target (*B*) and next instruction(*A*). The source and destination point to a finite set of memory cells holding 10-bit unsigned integers[4] branch and next fields point to other instructions, which live in a different (conceptual) address space. Our compiler represents both variables and instructions as labels.

Execution of a *movdbz* consists of fetching the value stored in the source cell and decrementing it. If the decrement succeeds, the value is stored in the destination cell and execution continues at A (the 'next instruction' field). If the decrement results in an underflow, 0 is stored in the destination field and execution continues at address B.

It has been proven that a strict subset of this instruction set, subtract-and-branch-if-negative (*SBN*) is Turing-complete in [7]. The *SBN* machine has a set of three-operand instructions (destination, source and branch target) and memory cells holding non-negative

[4]The values actually map to DWORD-aligned 32-bit pointers, which have to be valid and mapped stack pointers. The current prototype maps only a single page for the kernel stack, whereas this could easily be extended if more values need to be held

integers like our architecture, but as opposed to decrementing by a constant, every instruction subtracts the contents of the source from the destination operand and stores the result in the destination. If we can synthesize *SBN* from our *movdbz*; Assume we have an instruction `L: sbn X,Y, A, B //if(`$X - Y \geq 0$`) {X-=Y;goto A;} else {X-=Y;goto A;}` the following instructions are equivalent, where i1 through i3 are temporary instruction labels, tmp is an otherwise unused storage cell and INT_MAX is a storage cell initialized to 2^{10}. Note that the last 3 instructions are only needed to simulate the unsigned underflow discussed in the book, which is not strictly necessary for the proof of turing-completeness.

```
L: movdbz tmp, Y, i1, A
i1: movdbz X,X, L, overflow
overflow: movdbz X, INT_MAX, L, L
i2: movdbz tmp, Y, i3, B
i3: movdbz X,X, i2, overflow
```

Our compiler compiles each *movdbz* instruction entry into an IDT, a set of page tables and a TSS per instruction. Each memory cell is assigned a page of physical memory, with the value stored in the DWORD starting at offset 8 in that page. Furthermore, we have implemented a demonstration kernel that will initialize all the control registers to point to these tables without possible interference from the plethora of hardware a real kernel would initialize. The compiler also allows creating interrupt tasks pointing to valid pieces of code, so execution can be seamlessly transferred between *movdbz* code and regular X86 instructions, which our demo uses to fill the frame buffer with the results of the *movdbz* computation.

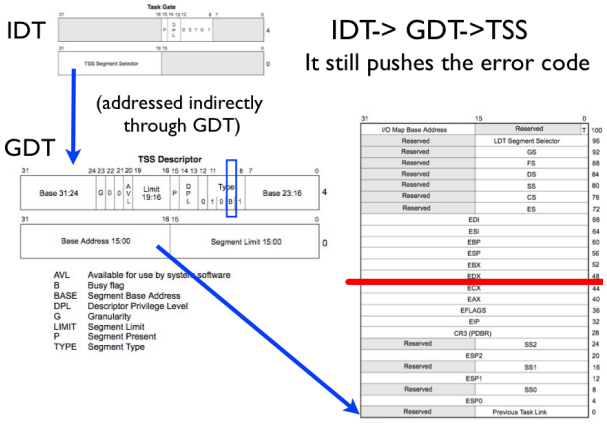

Figure 2: The flow of an interrupt, bits affecting handling highlighted. The TSS is split by a page boundary at the red line.

Each *movdbz* instruction is executed by either a page-fault or a double-fault. The conceptual 'rising clock' of

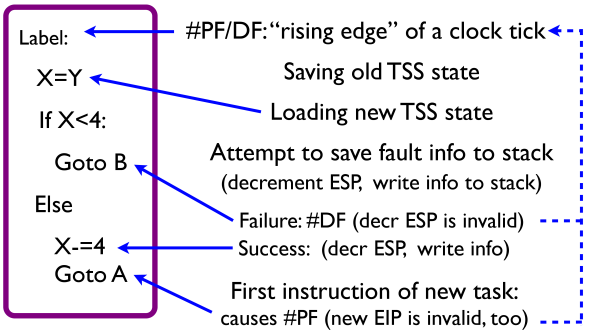

Figure 3: Elements of the *movdbz* instruction

our weird machine is the interrupt being raised. The CPU will walk the IDT and GDT to find the TSS selector of the instruction to be executed. This TSS selector will point to a TSS that is mapped across a page boundary, so the Task State Segment is split into two halves, as seen in Figure 2. Splitting the TSS across two pages allows two seperate areas of physical memory to be overwritten by one physical write, as described in Section 3.4.1. The page tables map the region in the upper page of this TSS to the physical page corresponding to the source memory address of the *movdbz* to be executed. The source cell's value (stored at offset 8 within that page) will be loaded as the interrupt handlers stack pointer from this page.

The lower part of the TSS is mapped to a page specific to this *movdbz* which contain the page tables for that instruction and the instruction pointer for the page-fault handler. One one hand, if we want to end our computation and return to normal CPU instruction, we can point this to valid code, effectively ending the chain of nested pagefaults. On the other hand, if we want to continue computing, we can point this to an invalid address so we will keep repeating faults.

The new page tables loaded as part of this TSS will remap the upper half of that TSS to point to the destination memory cell as opposed to the source cell. After loading the new page tables, the CPU will push an (ignored) error code to the stack pointer, thereby decrementing it by 4. As our stack pointers, i.e. memory cell values, are guaranteed by the compiler to fall within the lowest page, which we map, the push will succeed unless the variable was 0.

In that case, a double fault will be raised. Our compiler crafts the IDT and page tables so that the IDT entry for the double fault will point at the TSS belonging to the instruction referred to by the branch target in the current instruction. The upper half of the branch target's TSS will again be mapped to that *movdbz*'s source. Therefore, the CPU will save the current state, including the

zero it could not decrement further to the current TSS, setting the destination variable to 0 in the process and will resume 'execution' at the branch target *movdbz*.

However, if the push succeeded, the CPU will attempt to execute the (not mapped) interrupt handler and raise another page fault. The processor will save the decrement stack pointer to the memory cell as part of saving state, load the new TSS from the page fault IDT entry, which the compiler maps to the 'next instruction' *movdbz*. Hence, execution has been moved to the next instruction, the source memory cell of which has been loaded into the stack pointer.

3.4 Implementation Constraints

3.4.1 Busy bit

The interrupt task mechanism tries to specifically prevent looping interrupts, because the interrupt TSS contain back-links to the interrupted task. Furthermore, if a buggy system would accidentally enter infinitely looping interrupts, that situation would be hard to recover from or diagnose.

The loop-prevention mechanism hinges on the *busy bit* in the TSS descriptor in the GDT, which the CPU sets when entering a task and clears when leaving it. However, this simple automaton is not powerful enough to detect all loops, because its only state - the busy bit - resides in the same memory that the task switching mechanism operates on.

We notice that in addition to just reading and writing the stack pointer (which the interrupt handling logic will perform arithmetic on), the processor also loads and stores a number of general purpose registers without modification.

By placing a valid GDT descriptor into the lower half of the TSS when it is loaded and overlaying that page onto the GDT when saving state, we can overwrite the GDT entry with arbitrary values just after the busy bit has been set. In our case, we map EAX and ECX over the GDT entry and overwrite it with the exact same descriptor, but with the busy bit cleared. An even more obfuscated control flow could be implemented by writing a different GDT descriptor.

Furthermore the processor will reset if it encounters a triple-fault, i.e. another fault while handling a double fault. Therefore, the *movdbz* following a branch (after an underflow) must not cause another underflow. Typically, we work around this restriction by inserting another *movdbz* which writes a large positive value from a dummy memory cell to the memory cell that caused the underflow, simulating an unsigned underflow rolling over to a larger positive number.

3.5 Graph coloring

Another restriction on the task switching mechanism is that the process will only switch to a task other than the current task specified in the TR register. A trivial solution to this problem would be to use a different TR value for every *movdbz* instruction.

Unfortunately, because of the trick used to clear the busy bit described in Section 3.4.1, we can only use 16 different TR values that correspond to the 16 GDT slots just before a page boundary. We therefore have to map multiple *movdbz* Task State Segments to the same virtual address, so we need to restrict program flow such that no two *movdbz* at the same address follow each other.

When considering the *movdbz* instruction graph (where every node corresponds to an instruction and if $x.A = y$ or $x.B = y$, $(x, y) \in G$), the assignment of TSS slots is equivalent to 16 coloring the graph. If the particular instruction graph is not 16 colored, it can be easily extended by inserting semantically irrelevant instructions(decrementing an otherwise unused variable and storing it in another unused variable, then branching to the original branch target), i.e. subdividing edges on the instruction graph, until the graph is 16-colorable[5]

3.6 Evaluation

We released the source code for our *movdbz*compiler on GitHub.[6] Together with the compiler, we released a minimal kernel template that boots the processor, initializes the relevant registers to point at the compilers output and causes a page fault, beginning the interrupt based computation. Finally, we also include a demonstration implementation of the Game Of Life that demonstrates the ability to integrate real X86 code and interrupt based computation. The page-fault weird machine is used to compute each iteration of the automaton, using 31 *movdbz* instructions per cell. After each iteration, the CPU is restored to a valid state and normal X86 assembly instructions are used to output the cellular automatons configuration to the framebuffer. In principle, the size of the game of life is only limited by the available physical memory (which becomes a problem much earlier than execution time becomes an issue), however as described below the current compiler does not optimize for memory usage, effectively limiting programs to a few thousand instructions.

The current proof-of-concept compiler has some limitations that prevent practical exploitative use; All cre-

[5]This procedure terminates, because once every instruction i is preceded by one dummy instruction and succeeded by two dummy instructions, we can color i green, the predecessor blue and the successors red and yellow, coloring the graph trivially at the expense of quadrupling program size

[6]`https://github.com/jbangert/trapcc`

ated tables are encoded as C source code that places the appropriate values in memory using kernel-specific macros. While this approach is flexible, the emitted initialization code is quite large - the Game of Life demo is only demonstrated in small sizes because the initialization code hits limits in our bootloader and the C compiler. Furthermore, no effort was undertaken to minimize the memory usage of the weird machine - the two pages used for each TSS could for example be re-used for page tables and a single physical page can hold both the lower and the upper part of two distinct TSS's (effectively holding one instruction and one variable). If the pages are used in that fashion, the approximately 1 million physical memory pages on the 32-bit Intel architecture could each hold one instruction and one variable, not accounting for initialization code or memory mapped devices.

The current approach is also limited in its interaction with pre-existing kernel code; if the page-fault weird machine were to be used e.g. to hide a rootkit or to otherwise manipulate kernel structures, the current implementation can only change control flow and write 4 bytes on every page. Through compiler modifications (moving the page boundary within a TSS) a slightly larger fraction of memory could be used for arithmetic. Furthermore, the reloading and storing of TSS already moves values between memory locations as a side effect which could also be used to corrupt kernel structures. Ultimately though, the compiler would have to be extended by another primitive to allow arbitrary memory reads and writes[7].

4 Conclusion

Although address translation and memory access trapping subsystems of a modern processor are central to enforcing any and all practical security models, these systems per se are rarely studied in computational terms, as a unit and in composition with other features. However, since trustworthiness of computing systems is typically predicated on assurance regarding the types of computation these systems can and cannot be driven to perform, we believe that such a study of traps is long overdue.

Our result, taking inspiration from hacker research, demonstrates that unorthodox uses of MMU features in modern processes can result in powerful computations driven entirely by their in-memory descriptor tables – without dispatching any native instructions. We hope that our exposition of this "weird" sub-architecture will encourage follow-on research in enumerating and characterizing other unexpected computational environments in processors and chipsets.

[7] e.g. by loading a TSS from the middle of a page to read a value into the SP register and then faulting back to the Page-Fault computation

5 Acknowledgments

This research was supported in part by the Department of Energy under Award No. DE-OE0000097 and by Intel Lab's University Research Office. The views and opinions in this paper are those of the authors and do not necessarily reflect those of any of the sponsors.

References

[1] *Architectures Software Developer Manuals*. Intel Corporation, 2013.

[2] anonymous author. Once upon a free(). Phrack 57:9. `http://phrack.org/issues.html?issue=57&id=9`.

[3] Sergey Bratus. What Hacker Research Taught Me. In *RSS*, 2009. `http://www.cs.dartmouth.edu/~sergey/hc/rss-hacker-research.pdf`.

[4] Sergey Bratus, Michael E Locasto, Meredith L Patterson, Len Sassaman, and Anna Shubina. Exploit Programming: From Buffer Overflows to Weird Machines and Theory of Computation. *USENIX; login*, pages 13–21, 2011.

[5] Thomas Dullien. Exploitation and State Machines: Programming the Weird Machine, Revisited. In *Infiltrate*, April 2011. `http://www.immunityinc.com/infiltrate/2011/presentations/Fundamentals_of_exploitation_revisited.pdf`.

[6] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[7] William F. Gilreath and Phillip A. Laplante. *Computer Architecture: A Minimalist Approach*, chapter 6, page 41. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[8] Mario Heiderich. The innerHTML Apocalypse - How mXSS attacks change everything we believed we knew so far. In *SysCan conference*, 2013.

[9] Mario Heiderich, Marcus Niemietz, Felix Schuster, Thorsten Holz, and Jörg Schwenk. Scriptless Attacks: Stealing the Pie without Touching the Sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 760–771. ACM, 2012.

[10] jp. Advanced Doug Lea's malloc Exploits. Phrack 61:6. http://phrack.org/issues.html?issue=61&id=6.

[11] MaXX. Vudo malloc Tricks. Phrack 57:8. http://phrack.org/issues.html?issue=57&id=8.

[12] Hannes Mehnert and Andreas Bogk. A Domain-Specific Language for manipulation of binary data in Dylan. In *International Lisp Conference*, 2007. http://www.itu.dk/~hame/ilc07-final.pdf.

[13] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, Faster, Atronger SFI for the x86. In *PLDI*, pages 395–404, 2012.

[14] Nergal. The Advanced return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine*, 58(4), Dec 2001.

[15] James Oakley and Sergey Bratus. Exploiting the Ward-working DWARF: Trojan and Exploit Techniques with no Native Executable Code. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[16] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A Fistful of Red Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the 3rd USENIX conference on Offensive technologies*, pages 2–2. USENIX Association, 2009.

[17] PaX Team, http://pax.grsecurity.net/docs/pageexec.old.txt. *PAGEEXEC*, 2003.

[18] PaxTeam, http://pax.grsecurity.net/docs/pageexec.old.txt. *PAGEEXEC implementation*, 2000.

[19] Rebecca Shapiro. Programming Weird Machines with ELF Metadata. In *DEFCON 20*, 2012.

[20] Gerardo Richarte. Re: Future of Buffer Overflows. Bugtraq, October 2000. http://seclists.org/bugtraq/2000/Nov/32.

[21] Gerardo Richarte. About Exploits Writing. Technical report, Core Security Technologies, http://corelabs.coresecurity.com/index.php?module=Wiki&action=attachment&type=researcher&page=Gerardo_Richarte&file=publication%2FAbout_Exploits_Writing%2F2002.gera.About_Exploits_Writing.pdf, 2002. [Title as indicated.].

[22] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications, 2009.

[23] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Towards a Theory of Computer Insecurity: a Formal Language-Theoretic Approach. *IEEE Systems Journal, special issue Security and Privacy in Complex Systems*, 2011.

[24] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: return-into-libc without Function Calls. In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[25] Skape. LOCREATE: an Anagram for Relocate. *Uninformed*, 6, 2007. http://uninformed.org/?v=6&a=3&t=pdf.

[26] Sherri Sparks and Jamie Butler. ShadowWalker: Raising the Bar for Rootkit detection. In *DEFCON 13*, 2005.

[27] Joe Stewart. OllyBone: Semi-Automatic Unpacking on IA-32. In *DEFCON 14*, 2006.

[28] Yan Wang and Veronica Gaspes. A Library for Processing Ad hoc Data in Haskell : Embedding a Data Description Language. In *Implementation and Application of Functional Languages*, number 5836 in Lecture Notes in Computer Science, page 16, 2011.