



## Nonsense Code: A Nonmaterial Performance

Barry Rountree <routree\_at\_llnl\_dot\_gov>, Lawrence Livermore National Laboratory  <https://orcid.org/0000-0002-0087-4301>

William Condee <condee\_at\_ohio\_dot\_edu>, Ohio University  <https://orcid.org/0000-0003-3357-8485>

### Abstract

Critical Code Studies often relies on the textual representation of code in order to derive extra-textual significance, with less focus on how code performs and in what contexts. In this paper we analyze three case studies in which a literal reading of each program's code is effectively nonsense. In their performance, however, the programs generate meaning. To discern this meaning, we use the framework of nonmaterial performance (NMP), which is based on four tenets: code abstracts, code performs, code acts within a network, and code is vibrant. We begin with what is to our knowledge the oldest example of nonsense code: a program (now lost) from the 1950s that caused a Univac 1 computer to hum "Happy Birthday". Second, we critique Firestarter, a processor stress test from the Technical University of Dresden. Finally, we analyze one of the family of processor power side-channel attacks known collectively as Platypus. In each case, the text of the code is a wholly unreliable guide to its extra-textual significance. This paper builds on work in Critical Code Studies by bringing in methodologies from actor-network theory and political science, examining code from a performance-studies perspective and with expertise from computer science. Code can certainly be read as literature, but ultimately it is text written to be performed. Imagining and observing the performance forces the critic to engage with the code in its own network. The three examples we have chosen to critique here are outliers---very little code in the world is purposed to manipulate the physical machine. Nonsense shows us the opportunity that nonmaterial performance creates: to decenter text from privileged position and to recenter code as a performance.

## 1. Happy Birthday, Nonsense, and Nonmaterial Performance



**Figure 1.** Note that programming the UNIVAC 1 is done with pencil, paper, and flow charts.[Sperry Rand Corporation 1959]

Table 1. Sample of Univac instruction code C-10 (from author's files). The following is an excerpt from the documentation we received at EMCC to learn how to program. (In the instructions, *m* stands for memory.)

| Univac Instruction Code C-10 |             |  |
|------------------------------|-------------|--|
| Avg. time in microseconds    | Instruction | Description  |
| 525                          | Am          | Add ( <i>m</i> ) to ( <i>rA</i> )*, result in <i>rA</i> ; ( <i>m</i> ) also placed in <i>rX</i> .  |
| 445                          | Bm          | Clear ( <i>rA</i> ), then bring ( <i>m</i> ) into <i>rA</i> ; ( <i>m</i> ) also placed in <i>rX</i> .  |
| 445                          | Cm          | Place ( <i>rA</i> ) in <i>m</i> . Clear <i>rA</i> .  |
| 3,890                        | Dm          | Divide ( <i>m</i> ) by ( <i>rL</i> ) rounding off the quotient to 11 digits; result in <i>rA</i> . Unrounded quotient in <i>rX</i> .   |
| 445                          | Em          | Extract from ( <i>m</i> ) the characters (including digits) specified by ( <i>rF</i> ). Clear only those characters of <i>rA</i> which are replaced by the exacted characters. When a digit in <i>rF</i> is "0," leave the corresponding character in <i>rA</i> unaltered. When a digit in <i>rF</i> is "1," insert the corresponding character of <i>m</i> in <i>rA</i> . |
| 445                          | Fm          | Place ( <i>m</i> ) in <i>rF</i> .  |
| 445                          | Gm          | Place ( <i>rF</i> ) in <i>m</i> .  |
| 445                          | Hm          | Place ( <i>rA</i> ) in <i>m</i> without clearing <i>rA</i> (that is, Hold ( <i>rA</i> ) in <i>rA</i> ).  |
| 445                          | Jm          | Place ( <i>rX</i> ) in <i>m</i> ; ( <i>rX</i> ) unaltered.   |
| 285                          | Km          | Place ( <i>rA</i> ) in <i>rL</i> , clear <i>rA</i> ; disregard <i>m</i> .  |
| 445                          | Lm          | Place ( <i>m</i> ) in <i>rL</i> ; ( <i>m</i> ) also placed in <i>rX</i> .  |
| 2,150                        | Mm**        | Multiply ( <i>rL</i> ) by ( <i>m</i> ) rounding off the product to 11 digits; result in <i>rA</i> .  |
| 2,150                        | Nm**        | Negative Multiplication. Multiply ( <i>rL</i> ) by $-m$ rounding off the product to 11 digits; result in <i>rA</i> .   |

\* (*rA*) means "contents of register A," and so on.  
 \*\* Instructions M and N leave irrelevant information in *rX*.

Figure 2. Assembly language instructions for the UNIVAC 1.[Koss 2003]

Consider a somewhat mysterious computer tape from the early 1950s that was run a handful of times each year on Lawrence Livermore National Laboratory's (LLNL) UNIVAC 1 computer. The code is easy enough to describe. Single instructions are repeated thousands of times, but the results of those instructions are discarded. The instruction loops are occasionally repeated. There are only a couple dozen of these loops. There is no program input or output. Execution takes less than twenty seconds.

The giveaway is the tape label: Happy Birthday. The UNIVAC 1, which filled an entire room, was a noisy machine. The noise varied in pitch, and the pitch depended on the electrical consumption needed for the different individual instructions the machine happened to be running. Executing those instructions repeatedly would increase the duration of that particular tone. This program was a favorite of pioneering programmer Mary Ann Mansigh Karlsen, a physics coder at LLNL from the mid-1950s to the mid-1990s. She would help celebrate her colleagues' birthdays by calling their office from the machine room and serenading them by running the program on the tape [Rountree and Condee 2021]. Unfortunately, to the best of our knowledge, the tape for the UNIVAC 1 Happy Birthday has disappeared into history.

While hardly a consequential milestone in the history of computer science, this program does illustrate the limits of text-centric critical analysis. The semantic content of the code is irrelevant, yet that code is what drives the physical performance of the physical machine. Rerunning the code on a different computer (or a hardware emulator) reverts the performance back to nonsensical loops of instructions.

We offer this trivial example to illustrate a point central to this paper: code comprises text, purpose, and performance. That performance in turn is both material (the UNIVAC 1 humming Happy Birthday in the 1950s) and nonmaterial (mental models of a described, recollected, or intended performance). This code is the simplest example of what we define as nonsense code: the class of programs designed to change the physical state of the computer using side effects of the machine's instructions.

We begin with a short review of Nonmaterial Performance (NMP) and define nonsense code. We then describe Firestarter and Platypus as examples of nonsense code and critique them using the framework of NMP. Firestarter is a processor stress test that makes use of Intel's Running Average Power Limit (RAPL) technology, and Platypus is a set of processor side channel attacks that also make use of RAPL, as well as Intel's Software Guard Extensions (SGX). Both Firestarter and Platypus rely on manipulating the physical machine, and in both cases the code remains opaque

unless studied as performance.

Nonsense code poses challenges to traditional approaches in Critical Code Studies. We demonstrate how an NMP approach reveals how meaning, in the wild, can escape the authors' initial intent, even in nonsense code.

6

## 2. Nonmaterial Performance

Nonmaterial Performance (NMP) is a framework that uses performance studies, actor network theory, and vibrant matter to expose how code acts in the world. NMP is based on four tenets: code abstracts, code performs, code acts within a network, and code is vibrant. The performance of code comprises the mental models, the text, and the machine's physicality [Condee & Rountree 2020]. In this paper we bring the framework of NMP to bear on code that is profoundly physical.

7

### Code Abstracts

Abstractions, which are ubiquitous in computer science, hide nuance and detail in order to create concepts that are simpler to work with. There are dozens of layers of abstractions, beginning with the silicon, up through assembly language (the lowest level of abstraction available to programmers), and on to high-level programming languages and frameworks. These abstractions, of course, still affect the machine at the physical layer — the abstractions hide complexity, not remove it.

8

There is a similar idea of abstraction in mathematics, but mathematical abstractions such as `sum()` and `average()` destroy information (one can't get back to the original data given a sum or average). Abstraction in computer science only hides the information [Colburn & Shute 2007]; underneath the assembly language instructions, the picky details still exist. Abstraction in computer science, then, is done for the convenience of the programmer.

9

If one wants to manipulate the physical machine through code, the only tools available are the higher-level abstractions of programming languages. Given a deep enough knowledge of a particular architecture and software stack, one can use these abstractions to change the properties of the physical machine. Because the abstractions were not created to bring about physical change, the resulting code looks like nonsense.

10

### Code Performs

The textual representation of code, like the script of a play, is only a score for its performance. According to Richard Schechner, the focus of performance studies is human behavior: "any action that is framed, presented, highlighted or displayed is a performance" [Schechner 2006, 2]. Performance studies provides useful models for highlighting behaviors not previously framed as performance and/or not privileged as objects for study, including not only the arts, but also sports, business, sex, ritual, play, everyday life — and technology (although Schechner acknowledges that the last is "not usually analyzed") [Schechner 2006, 31].

11

Schechner famously defines performance as "restored behavior" and "twice-behaved behavior," and goes on to suggest that when "oneness" is "broken down finely enough and analyzed," the behaviors "are revealed as restored behaviors" [Schechner 2006, 29]. In other words, it is restored behavior all the way down. Similarly, in computing, individual actions (in the form of assembly language instructions) are all restored behavior — there is nothing new that can be done on a processor. It is only the aggregation of twice-behaved behavior that makes novelty possible. Both programmers and performers wield their creative power through the combination of these existing behaviors [Rountree and Condee 2021, 304–5]. Code Acts within a Network

12

Code performs within multiple layers of networks, comprising both objects and humans. NMP employs actor-network theory (ANT) to crack open computing in ways not possible with standard computer science approaches. From an ANT perspective, everything in the social and natural world is the result of a diverse web of "materially heterogeneous" relations [Law 2009, 143]. ANT privileges neither humans nor objects, since neither alone determines the processes and outcomes of these networks. Instead, networks consist of the interaction of humans and objects, with implications for all. ANT uses Bruno Latour's term "actant," which Jane Bennett defines as "a source of action that can be either

13

human or nonhuman; it is that which has efficacy, can do things, has sufficient coherence to make a difference, produce effects, alter the course of events” ([Bennett 2010, viii, emphasis in original]; [Rountree and Condee 2021, 302]). Code is Vibrant

We borrow the idea of vibrancy from Bennett’s book, *Vibrant Matter: A Political Ecology of Things*. While matter is conventionally viewed as passive, Bennett asserts that matter is vibrant. She describes the “vitality” of matter as “the capacity of things . . . not only to impede or block the will and designs of humans but also to act as quasi agents or forces with trajectories, propensities, or tendencies of their own” [Bennett 2010, viii]. Her goal is “to articulate a vibrant materiality that runs alongside and inside humans” [Bennett 2010, viii]. Similarly, the conventional view of code requires that it be inert: fashioned to fulfill the single intent of its author. From an NMP perspective, code in performance also has vibrancy: the ability to act and have influence independent of its creators [Condee & Rountree 2020, 149]. As we will conclude, what code execution means ultimately accrues in the domain of human perception, and these meanings are informed by the unseen ensemble of performance occurring behind the screen. That meaning is unstable, independent of the author’s intention, and contingent on time and place. NMP makes the nonmaterial visible [Condee & Rountree 2020, 303–4].

14

### 3. Nonsense Code

As noted above, we define nonsense code as the class of programs designed to change the physical state of the computer using side effects of the machine’s instructions. The text of code abstracts away physical details such as energy consumption, thermal characteristics, and electrical resonance; nonsense code uses its text to manipulate those physical characteristics. Examining this class of code without considering its performance — what it actually does to the physical machine — renders it nonsense. In the case of *Happy Birthday*, the machine instructions of the UNIVAC 1 did not provide a mechanism for creating sounds. That physical effect had been abstracted from the text of the instructions. What makes the *Happy Birthday* tape compelling is that the programmers ignored the instructions’ semantics (for example, adding up numbers) in order to manipulate the material implementations of those instructions (for example, making the machine hum the note “G”). The result is textual nonsense and an amusing performance. Assembly language instructions will, in performance, inadvertently cause a computer to consume more power, increase its temperature, and resonate at a particular frequency. The text of these instructions, however, does not convey this physicality. Nonsense code, then, relies on the physical side effects that have been abstracted away from the code.

15

Evolvable hardware (and the evolutionary algorithms used to create it) bears more than a passing resemblance to nonsense code: the results are often impenetrable, but the reason for the impenetrability differs. The evolutionary algorithms used for creating hardware manipulate theoretical abstractions of actual physical components to arrive at a solution that fits the constraints of the problem. That abstract solution can then be transformed into physical hardware [Cancare et al. 2011]. Nonsense code also uses abstractions, but does so in order to manipulate the physical properties of a machine that does not appear in the abstractions. The evolutionary algorithm for evolvable hardware does not know and cannot discover what details have been elided in the abstractions it was given. Those details are available to the author of nonsense code.

16

Most examples of Critical Code Studies focus primarily on the textual representation of a program, which is the immediate point of entry for code analysis. Our approach centers instead on the performance of code, which is especially valuable for the vast majority of cases for which text is unavailable. Code may be unreleased (*Platypus*), proprietary (*RAPL*, *SGX*), or simply lost (*Happy Birthday*). In such cases, approaches from performance studies, relying on paratextual information, including oral history, documentation, specifications, and scholarship, restore and interpret ephemeral performances.

17

### 4. Firestarter

**Author:** Daniel Hackenberg

**Citation:** D. Hackenberg, R. Oldenburg, D. Molka and R. Schöne, “Introducing Firestarter: A processor stress test utility,” 2013 International Green Computing Conference Proceedings, Arlington, VA, USA, 2013,

**Version:** 1.7.4 (1.0 released in 2013, 2.0 released in 2021)

**Source:** <https://github.com/tud-zih-energy/Firestarter>

**Invocation:** `./Firestarter - timeout=60 -report`

**Technical Description:** Firestarter is a processor stress test, essentially a program that makes the processor work as hard as possible in order to determine how the processor will behave under high load, for example, testing cooling systems at maximum power [Hackenberg et al. 2013]. Prior to the creation of Firestarter, best practice was to use numerically intensive codes such as Prime95 or LINPACK. These codes, however, were written to solve mathematical problems, and only incidentally required a large amount of power. Firestarter is designed to consume maximum power — and nothing else. As such it can reach both higher and more consistent levels of power consumption, thereby producing more reliable results. Firestarter's code is a carefully constructed mass of assembly language instructions that work nearly all of the components of the processor simultaneously and as hard as possible — without calculating anything useful at all. Firestarter thus meets our definition of nonsense: its purpose is to change the state of the physical processor using the side effects of the available assembly language instructions.

In this section we briefly critique Firestarter, as well as demonstrate its vibrancy when repurposed.

```
1 FIRESTARTER - A Processor Stress Test Utility, Version 1.7.4 (github), build: 2021-04-14
2 Copyright (C) 2019 TU Dresden, Center for Information Services and High Performance Computing
3 This program comes with ABSOLUTELY NO WARRANTY; for details run `FIRESTARTER -w`.
4 This is free software, and you are welcome to redistribute it
5 under certain conditions; run `FIRESTARTER -c` for details.
6
7 system summary:
8 number of processors: 2
9 number of cores per package: 28
10 number of threads per core: 2
11 total number of threads: 112
12
13 processor characteristics:
14 architecture: x86_64
15 vendor: GenuineIntel
16 processor-name: Intel(R) Xeon(R) Platinum 8276L CPU @ 2.20GHz
17 model: Family 6, Model 85, Stepping 7
18 frequency: 2199 MHz
19 supported features:
20 - x86_64 FPU MMX SSE SSE2 SSE3 SSSE3 SSE4.1 SSE4.2 POPCNT AVX AVX2 AVX512 FMA AES SMT
21 Caches:
22 - Level 1 Data Cache, 32 KiB, 8-way set associative, shared among 2 threads
23 - Level 1 Instruction Cache, 32 KiB, 8-way set associative, shared among 2 threads
24 - Unified Level 2 Cache, 1024 KiB, 16-way set associative, shared among 2 threads
25 - Unified Level 3 Cache, 39424 KiB, 11-way set associative, shared among 56 threads
26
27 Taking AVX512 path optimized for Skylake-SP - 2 thread(s) per core
28 Used buffersizes per thread:
29 - L1-Cache: 16384 Bytes
30 - L2-Cache: 524288 Bytes
31 - L3-Cache: 720896 Bytes
32 - Memory: 524288000 Bytes
33
34 using 112 threads
35 - Thread 0 runs on CPU 0, core 0 in package: 0
36 - Thread 1 runs on CPU 1, core 1 in package: 0
37 ....
38 - Thread 110 runs on CPU 110, core 29 in package: 1
39 - Thread 111 runs on CPU 111, core 30 in package: 1
40 The cores are numbered using the IDs from sysfs (see sys/devices/system/cpu/
41 cpu-no./topology/) or /proc/cpuinfo. These IDs do not have to be consecutive.
42
43
44 performance report:
45
46 Thread 0: 111620885 iterations, tsc_delta: 132028983926
47 Thread 1: 111893575 iterations, tsc_delta: 132000083244
48 ....
49 Thread 110: 116392352 iterations, tsc_delta: 132001907870
50 Thread 111: 116305602 iterations, tsc_delta: 132001990456
51
52 total iterations: 120632625799
53 runtime: 60.02 seconds (132031227082 cycles)
54 estimated floating point performance: 2297.37 GFLOPS
55 estimated memory bandwidth: 96.01 GB/s
56 # this estimate is highly unreliable if --function is used in order to select
57 a function that is not optimized for your architecture, or if FIRESTARTER is
58 executed on an unsupported architecture!
```

Figure 3. A brief Firestarter run on 112 CPUs across two Xeon processors.[Rountree and Condee 2021]

We begin with the output of a short Firestarter run (Figure 3). A few technical details require explanation in order to understand how this code becomes nonsense. Lines 1-5 are code attribution and licensing information. Lines 7-41

describe Firestarter’s understanding of the architecture it is running on. For clarity, we have included the results of only four of the 112 CPUs: 0, 1, 110 and 111. The performance report begins on line 44 and provides a total number of iterations completed for each CPU during the run (again, we only show CPUs 0, 1, 110 and 111). Adjacent to those values is the amount of time taken in clock ticks (`tsc_delta`).

In the performance recorded in Figure 3, Firestarter is executed for 60 seconds across 112 hyperthreads on two Cascade Lake processors. Lines 46-50 are of particular interest. Each thread reports how many iterations of the Firestarter payload loop it executed and how long it spent running those loops. The elapsed time for each thread is remarkably consistent: the difference between the longest-running thread of the four threads shown (thread 0) and the shortest-running thread (thread 110) is just over 0.02%. 21

The total work accomplished, however, by thread 110 is 4.2% greater than thread 0. Under this kind of load and this particular power management regime, not all hyperthreads are created equal (we will return to this phenomenon momentarily). 22

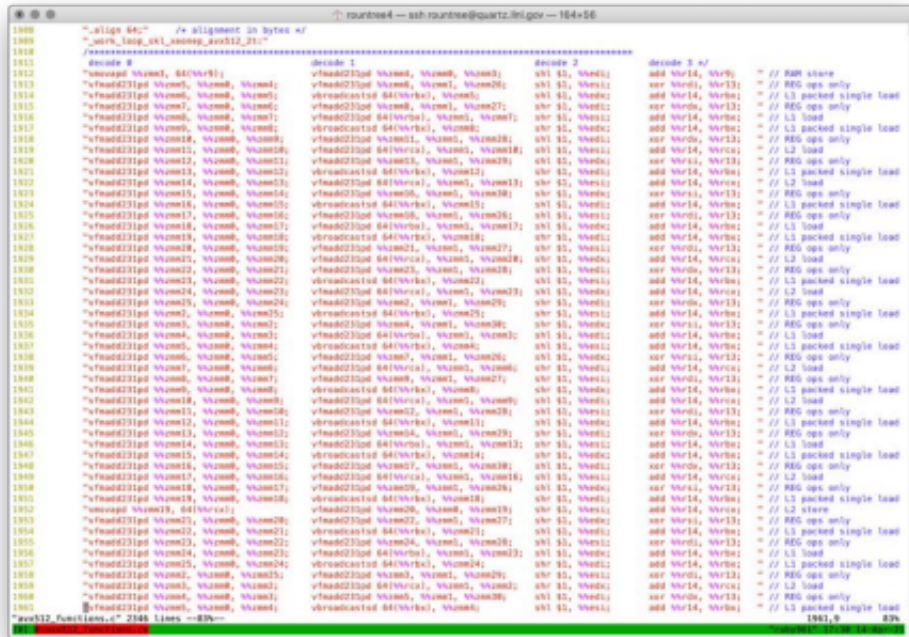


Figure 4. A portion of the handwritten assembly code used during the Firestarter execution recorded in Figure 3 [Hackenberg et al. 2013].

Figure 4 shows a portion of the code being executed. The code is x86 assembly language using AT&T syntax embedded in a larger C language function. There are two unusual aspects of this code. First is the four-column formatting, since each CPU in this architecture can retire up to four instructions per cycle — in simpler terms, each CPU can do four things at once. After a great deal of study and experimentation, the Firestarter authors were able to handcraft the selection and placement of these instructions so that they do in fact execute simultaneously. As a result, the program causes the computer to do more work and consume more energy per unit of time. Their accomplishment may not appear at first blush to be that astonishing: each CPU can execute four instructions, and they wrote code to do that. However, most non-nonsensical programs rarely have more than one instruction executing at a time, and very few reliably get up to three instructions per tick. What the Firestarter authors accomplished was to assemble a jigsaw puzzle in space and time, finding exactly which instructions could be cobbled together based only on their particular fit. This approach exposes the second unusual aspect of Firestarter: it uses a great deal of energy to do nothing in particular. 23

Closer study of the code reveals multiple duplicated patterns: results are repeatedly calculated, overwritten, and discarded. The simplest pattern occurs in the instructions placed in the “decode 2” column. 24

```
decode 2
shl $1, %%edi;
shl $1, %%esi;
shl $1, %%edx;
shr $1, %%edi;
shr $1, %%esi;
shr $1, %%edx;
```

Figure 5. Detail of Figure 4 [Hackenberg et al. 2013].

The values in these three registers (*edi*, *esi*, and *edx*), as seen in Figure 5, are shifted to the left by one bit (*shl*), then shifted back to the right by one bit (*shr*), over and over again. At a semantic level, this repeated shuffling back-and-forth accomplishes nothing. Moving the values in these 64-bit registers does not consume nearly as much power as the 512-bit registers used in the “decode 0” and “decode 1” columns, nor does it take as much power as the cache transfers under “decode 3.” But these shift instructions do soak up that last bit of execution capacity, thus they have found their place here.

25

## Critique

Focusing solely on the text of the code itself runs headlong into its pointlessness. Alternatively, including the authors’ intent (as recorded in the code comments and authors’ published papers) reveals one meaning of Firestarter, but does not exhaust the potential for other meanings. Code is sufficiently vibrant to escape the intent of the authors, and we now discuss how the nonsense of Firestarter reveals the messy details behind another abstraction: all processor cores are created equal.

26

In introducing Firestarter, we touched on its origins in heating and cooling processors. Most cooling occurs when the processor decides to run more slowly (thus consuming less energy and generating less heat), so as not to cook itself. To understand how this relates to Firestarter, it is necessary to go back to August, 2008, and Intel’s introduction of the multicore Nehalem processor architecture.

27

Prior to Nehalem, processors had a maximum speed (the CPU clock frequency) that would support any workload. Nehalem changed that equation. Restricting a workload to a single core meant that particular core could run at the maximum clock frequency; using more than one core lowered the ceiling for all cores. Higher clock frequencies use quadratically more power, so for a given processor, the programmer could spend the power budget on fewer, faster cores or more, slower cores. For the first time, configuration had to encompass both code and the underlying processor.

28

In subsequent architectures, power and performance management strategies included in the processor became increasingly sophisticated, and while manufacturers exposed a few interfaces to this firmware, processor vendors kept the underlying actants (algorithms and implementation) opaque. A program like Firestarter allowed researchers to see how processors reacted to predictable loads, particularly those that were designed to draw maximum power and generate maximum heat.

29

As an example, here are the results of multiple Firestarter runs on the dual Cascade Lake machine mentioned above.

30



| Number of cores | Units of work completed in 60 seconds |
|-----------------|---------------------------------------|
| 1               | 1.0                                   |
| 14              | 9.9                                   |
| 28              | 13.3                                  |
| 56              | 27.0                                  |
| 112             | 24.7                                  |

**Table 1.** Firestarter 60-second runs, using different numbers of cores (column 1). The resulting units of work completed (column 2) scale poorly. (Rountree)

Here, the use of Firestarter has shifted. Rountree repurposed an unmodified Firestarter into a performance evaluation tool. A single CPU completes a (normalized) unit of work in 60 seconds. Increasing the number of CPUs running simultaneously from 1 to 14 does not result in 14 units of work completed; instead, the result is just 9.9. Using all 112 CPUs only increases the completed work to 24.7. The fact that this processor scales poorly is less relevant here than Firestarter being used in a novel way: as a generic work generator. The purpose is malleable, even if the text is fixed.

31

Table 1 (above) shows a small example of using a nonsense code for a purpose other than for which it was designed. While one can delineate code through its purpose, text, and performance, doing so does not prevent code from being repurposed and re-presented. Because of this potential to be repurposed, code is vibrant. Code is not defined by a single meaning; it generates multiple meanings, and understanding one particular meaning does not necessarily give insight into any others.

32

Earlier in this paper we noted the different magnitudes in time and work completed across different hyperthreads (Figure 3 lines 46-50). Firestarter had been developed on a handful of machines, and any differences across cores, processors, or threads were not seen as significant enough to include in the initial paper introducing Firestarter. Rountree ran Firestarter across 4,200 Broadwell processors for 350 minutes (60 seconds for each run). Across that much larger population of processors, the observed variation became much more interesting.

33



Variation within three Broadwell processors (best, median, worst of 4200)

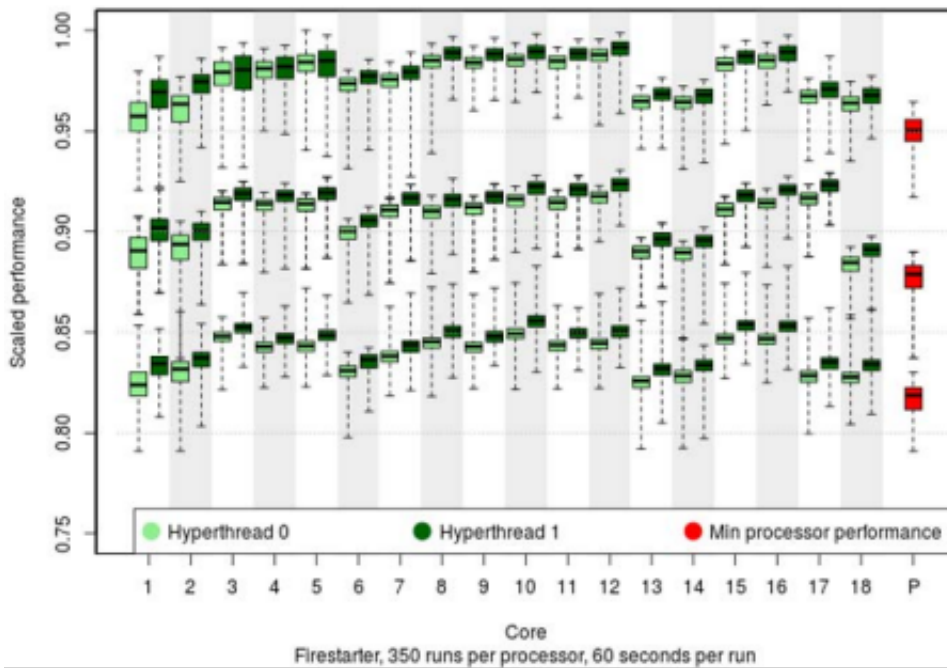


Figure 6. The three bands of box-and-whisker plots are the best, median, and worst out of 4,200 Broadwell processors. Note that the variation within this single processor model can exceed 20% [Marathe et al. 2017].

The three horizontal bands in Figure 6 correspond to the best, median, and worst processors out of the 4,200 processors characterized. These Broadwell processors have 18 cores with two hyperthreads per core. Each box-and-whisker plot represents the scaled number of iterations for a single hyperthread across 350 60-second executions of Firestarter (the box represents 50% of the results, the whiskers represent the total range).

34

The most striking result from this graph is that the performance of the best, median, and worst processor under Firestarter do not overlap at all, despite being ostensibly the same processor model. There is also variation across cores (core 12 is consistently better than 13), hyperthreads (hyperthread 1 is consistently a bit better than hyperthread 0 across all cores and processors), and from run to run (the height of each box-and-whisker plot). What Firestarter reveals, serendipitously, is the nuance and detail abstracted away by the shared concepts of “processor,” “core,” and “hyperthread.” For a particular processor model, the received abstraction is that all processors of that type are identical, certainly insofar as performance is concerned. Within a given processor, a core is abstracted to the point where it becomes identical to all other cores, and likewise for hyperthreads; there is no programmatic method for distinguishing them. Slight variations within the silicon will make some processors more or less efficient — essentially, it takes fractionally more energy to push bits down the wire. Processors that, by luck of the draw, are less efficient will heat up more quickly at the same CPU clock speed, meaning they will have to slow down sooner, and thus get less work done. Note that Firestarter remains unmodified and the purpose has shifted yet again: Firestarter is now a tool for revealing variation in silicon across populations of processors. The only way of getting to the nuance and detail of processor performance is through observing that performance under load. And the only way of maximizing that load is through nonsense.

35

From some previous Critical Code Studies perspectives, the starting point is the text of Firestarter, and while the comments and apparatus surrounding that code do indicate the authors’ intentions, textual analysis does not exhaust the potential meanings. Firestarter (and nonsense codes in general) generate meaning based on performance in particular environments. NMP centers the analysis on the performance.

36

Returning to the tenets of NMP described above (code abstracts, code performs, code acts within a network, and code is vibrant), we focus here on how Firestarter is vibrant. To make the notion of vibrancy more concrete, consider

37

puppetry. Shari Lewis, the great puppeteer of Lamb

Chop fame, describes the importance of discovering “what the puppet wants to do” (quoted in [Bell 2008, 7]). John Bell goes on to describe the “weird concept of letting the object determine action” [Bell 2008, 7]. A puppeteer may design a puppet for a particular character, play, and action. Through play, however, the puppet reveals what it can do and what it wants to do, unveiling a myriad of new possibilities. In other words, the puppet has vibrancy.

38

Programmers (and perhaps code critics) can be tempted to think that code does one thing. Instead, playing with Firestarter reveals what else it wanted to do: expose inhomogeneity [Inadomi et al. 2015]. Rountree allowed Firestarter to perform in new ways, which was unplanned by its authors. Because code is vibrant, those actions were discoverable and discovered. Firestarter reveals that, even at the level of assembly language, individual instructions have their own vibrancy. “Shift-left” (*shl*) does indeed shift the bits in the given register to the left, but it also takes time, draws power, fills a decoder slot, affects how surrounding instructions are scheduled by the processor, and can ultimately slow the processor down and unmask variation. None of this potential vibrancy is recorded in the textual representation of “shl.” Accessing that vibrancy requires the combination of an imagined, nonmaterial performance and the careful measurement of a material performance.

39

Firestarter illustrates the pointlessness of trying to nail code that has been released in the wild to a single meaning or purpose. Firestarter was explicitly designed as the epitome of pointless busywork. Without modification, however, it was used to measure work rates under increasing loads and core counts, and as a tool for quantifying processor, core, hyperthread, and runtime variation across thousands of processors.

40

The skilled puppeteer asks, “What does the object want (and not want) to do?” Forcing thousands of processors to behave identically is possible, but only by limiting their performance to the lowest common denominator. The notion of vibrancy enables the question, “What is possible given that supercomputers consist of thousands of inhomogeneous processors?” Once that question is on the table, computer scientists can rethink job scheduling, power efficiency, and performance reproducibility. From the Critical Code Studies perspective, vibrancy opens the door to thinking about processors as individual, unique entities whose traits (e.g., efficiency, speed, thermal characteristics, etc.) exist across a spectrum.

41

## 5. Platypus

**Author:** Moritz Lipp et al.

**Citation:** “PLATYPUS: Software-based Power Side-Channel Attacks on x86”, 2021 IEEE Symposium on Security and Privacy.

**URL:** <https://platypusattack.com>

**Source:** Proof-of-concept code has not been released.

## Safecrackers and Side-Channel Attacks



Figure 7. Headline from 1950s newspaper article about safecracking: “Locksmiths Used to Worry About Yeggs; Now It’s Spies”[Winget 1950].

Daniel Gruss and his students at the Technical University of Graz created Platypus to demonstrate the feasibility of one form of side-channel attacks that exploit processor vulnerabilities. Gruss is best known for co-discovering and analyzing the Spectre [Kocher et al. 2019] and Meltdown [Lipp et al. 2018] side-channel attacks. In these attacks, information designed to be kept secret on a computer was attacked using the processor cache as a side-channel. For Platypus, the side channel is the processor’s energy accounting system. These kinds of attacks leverage features found in nearly all

42

mobile, consumer, and server-grade processors. They are particularly difficult to guard against because they exploit the underlying system architecture rather than software faults.

Platypus was announced to the world in November, 2020 ([Cimpanu 2020]; [Peterson 2020]). Shortly thereafter, processor-power research was restricted at LLNL while the seriousness of the vulnerability was evaluated by a team headed by Rountree, as well as by teams at Intel, AMD, IBM, nVidia, and Arm. As a result, Intel updated its processor microcode and “a Platypus attack” entered the security lexicon.

43

To understand side-channel attacks, consider state-of-the-art safecracking in 1950: Combination locks are simple. Discs with slots are aligned until a lever falls into the slots. Then the lock opens. The moving discs and the falling lever make a noise. Legendary cracksmen filed their fingertips to the quick and felt the movement. So locks were refined. Then cracksmen used stethoscopes to listen to the movement. Locks were then made too smooth for that device.

But the war brought on the development of electronic detection of supersonic sound. In any radio store you can buy the apparatus. Cracksmen use an aerial the size of a knitting needle some six inches high set in a base the size of a biscuit. Push it against the safe dial. The sound is picked up, amplified in a box about six by four by two inches in size and recorded on a dial like the ammeter in a car.

What happens inside the combination lock is read as easily as an electro cardiograph by a physician. Which means it is not easy, but it can be done by an expert. [Winget 1950]

The author quoted above offers an abstraction of a combination lock: “[d]iscs with slots are aligned until a lever falls into the slots.” When this abstraction is translated into physical reality, a side-channel opens: metal rubs against metal, levers scrape along dials, and information leaks. Side-channel attacks, then, are not head-on: no dynamite blows open the safe door. Instead, the side-channel attack relies on details of the physical machine that have been abstracted away.

44

## SGX and the Cloud

The particular Platypus attack we critique recovers a cryptographic key from Intel’s Software Guard Extensions system. SGX, as it is more commonly known, was introduced in 2013 as the most recent attempt to solve a problem of great commercial interest: how can a user safely run code on a machine owned and maintained by someone they do not necessarily trust? For example, a hypothetical tech startup has a new way of applying machine learning to medical imaging. The images can be encrypted at the hospital and sent to the startup and the results can be encrypted and sent back. But the startup doesn’t want to own the pile of computers needed for the analysis when it is so much cheaper to rent them from Amazon, Google, Microsoft, or another cloud provider.

45

SGX is yet another layer of defense to address the scenario in which an attacker has gained control of the machine and can observe and modify the operating system. SGX adds the capability to create what is called an enclave: a secure space that would allow code written and uploaded by the medical imaging startup to be executed in such a way that no compromised operating system can read it. The startup can cryptographically sign their code before loading it, and then have SGX validate that it received the code unmodified. Even if an adversary has physical possession of the machine and the ability to tap the memory bus, no decrypted information will be visible.

46

In this example, the startup would put code in the enclave to decrypt the incoming image, do whatever it does with machine learning, and encrypt the results. The small amount of code that runs outside of the enclave only forwards encrypted images into the enclave and forwards encrypted results back to the hospital.

47

Platypus managed to crack this system. To see how this was done, we need to sketch two more subsystems in Intel processors: Running Average Power Limit (RAPL) and the Advanced Programmable Interrupt Controller (APIC).

48

## RAPL

For Platypus, energy measurement is the side-channel capable of being attacked. In response to the needs of both its

49

mobile and datacenter customers, Intel unveiled its Running Average Power Limit (RAPL) technology in its Sandy Bridge architecture in 2010. RAPL allows the operating system to set upper limits on the amount of power the processor is allowed to consume and, more crucially for Platypus, measure how much energy has been used. Every so often (in this case, approximately every fifty microseconds), the energy meter on the processor is updated: a particular register is incremented by the number of fractional Joules that have been consumed since the last update.

Given physical access to a much simpler (and slower) processor and an expensive oscilloscope, one can surreptitiously observe the electrical signal generated by individual instructions, and perhaps even infer something about the data. The general term for this kind of attack is Differential Power Analysis [Kocher, Jaffe, and Jun 1999]. Given a large number of electrical traces at a high enough sampling rate, good-enough guesses can be made about enough of the cryptographic key to allow its eventual recovery. For simple systems that are expected to be in the physical control of an adversary (smartcards, mobile devices), this analysis is a realistic method of attack. What appears to preclude its use here in more complex processors is that thousands of instructions can be executed over the fifty-microsecond energy update time. There is not enough energy information to identify individual instructions, thereby making this attack apparently infeasible in the real world.

50

## APIC

The second Intel subsystem required by Platypus provides precise user control over interrupts. The Advanced Programmable Interrupt Controller (APIC) is a much older technology with roots in the 80486 Intel processor introduced in 1989. The idea of interrupts goes back much further (to the 1953 UNIVAC 1103), solving what was then a universal performance problem. Prior to interrupts, when a program needed to interact with the outside world (for example, reading or writing to a tape), all progress came to a halt until that read/write request succeeded or failed. Interrupts allowed a program to do two things at once: the program, while continuing to run, says, in effect, “Go write this data to the tape drive and interrupt me when you’re done.” While the program is running, the tape device can complete its operation in its own time. When the tape completes, it notifies the program by raising an interrupt.

51

At the physical level, when the processor receives an interrupt, whatever code happened to be running is interrupted, and control is transferred to the operating system, which checks for errors and updates needed bookkeeping. Once that is complete, the previously running code is allowed to continue (usually none the wiser that it had been momentarily halted).

52

APIC allows the operating system to trigger an interrupt at a precise moment in the future. There are several registers that keep track of time. At each clock tick, 1 is added (incremented) to the current value in the register. But rather than incrementing once per second, recent processors increment much faster — more than one billion times each second. Assembly language instructions in a program take several clock ticks (billionths of a second) to make it through the processor pipeline, so this clock is essentially running faster than instructions can execute. In a bit of foreshadowing, APIC can set an interrupt to occur just a handful of ticks in the future.

53

We now have all the pieces in place to describe the Platypus attack.

54

## The Attack

Returning to the attack on the enclave: What happens when an interrupt is received when code is executing in the secure space of an SGX enclave? There are two bad choices: First, put off dealing with the interrupt until the secret code that is executing in the secure enclave completes (which might be anywhere from milliseconds to weeks). Second, hand off control to the operating system and hope it has not been compromised. SGX makes a better choice. SGX wipes out any unencrypted data in the cache and only then transfers control to the (untrusted) operating system. The operating system sees no data, encrypted or otherwise, in the cache and cannot decipher any of the encrypted data still in memory. When the operating system finishes handling the interrupt, it passes control back to SGX, which will reload the enclave back into cache and decrypt the secret code again. An attacker who is able to generate interrupts at will, for example through APIC, is still not going to get a chance to see unencrypted data. So far, SGX appears secure.

55

How does one break into this locked room? During that interrupt-handling process, the side channel has continued to accumulate energy. Can the measurements reveal what was happening during enclave execution? Not really. At best, that measurement refers to the accumulated energy of several thousands of instructions that executed during the fifty-microsecond update window. There's no way to pick out individual instructions from that single energy reading, much less their associated data. SGX still appears to be secure. The technique used by Platypus to solve this locked-room mystery is called zero stepping. Debuggers allow programmers to slowly single-step through each assembly language instruction to observe its effects at human timescales. Zero-stepping, in contrast, executes the same single instruction over and over again. If a secret instruction running in an enclave could be zero-stepped for fifty microseconds, the energy measurement would cover only that instruction (and the interrupt process). That measurement may be enough to identify the instruction and its data. Processors do not support native zero-stepping, but processors do provide APIC. Platypus's trick is to schedule the APIC interrupt a handful of ticks in the future to throw an interrupt just as the first secret instruction running in the enclave is finishing. Platypus has already added code (an "interrupt handler") into the compromised operating system to schedule another interrupt the same number of ticks into the future as the first. The effect is that the first secret instruction in the enclave is executed over and over again, never quite finishing, because it keeps getting interrupted. If this pattern can be held for a hundred microseconds or so, the energy measurement will capture that instruction, as well as all the other instructions involved with firing the interrupt, running the interrupt handler, and restarting the enclave. When enough energy information has been gathered on the first secret instruction, a similar sequence begins: the second instruction is executed over and over again. The net effect is that Platypus gets precise-enough energy measurement on each secret instruction executing in the secure enclave to decode it. That is enough to identify not only the instruction but the data it uses. Platypus effectively creates an oscilloscope out of a combination of APIC and RAPL, thereby recovering the secret key used by the enclave to decrypt the secret program.

Absent all the above context, the execution of a Platypus attack is inscrutable: a single instruction is continuously interrupted over and over and over again. That process of continuous interruption modifies the underlying physical state of the machine. Measuring that state allows the recovery of secret information. Platypus is nonsense.

A noncomputing example might elucidate how Platypus works, albeit in a simplistic and partial way. Bill obsessively vacuums his bedroom. Barry is aware of this hangup and is trying to help Bill. So, Bill starts vacuuming, and Barry knocks on the door. Bill, trying to mask his behavior, stops vacuuming, hides the Hoover in the closet, and invites Barry in. Barry looks around, sees nothing, and leaves. Bill resumes cleaning, Barry knocks again, Bill hides his Hoover, Barry enters, looks around, and leaves. Meanwhile, Bill has a few lights on (he's not a Roomba and can't vacuum in the dark) and is making coffee (Bill's obsession runs on caffeine). This cycle of vacuuming, knocking, hiding, entering, and leaving repeats scores or hundreds (or in the case of Platypus, thousands) of times. Barry, still suspicious and concerned about his friend, proceeds to examine Bill's electrical meter. He is able to read the background usage, indicating the ongoing use of lights and Mr. Coffee. But he is also able to see repeated spikes of electricity usage of a particular shape. Because Barry owns a similar vacuum cleaner, he knows the particular electrical pattern of Bill's WindTunnel Air Steerable Upright (as opposed to the spikes from the dryer or Bill's welding hobby). Busted. Reading the electrical meter is a side channel attack by which Bill's activity can be covertly observed and disambiguated. And, to return to Rountree's story, this is what caused LLNL to, metaphorically, lock up their power meters.

Returning to the startup example: the attacker knows the first task the enclave is going to perform is decrypt the data. Determining which encryption algorithm is used, how long the key is, etc., will be a tedious effort, but it's an effort that can be automated. At the end of the process, the attacker has the key and can decrypt the medical images prior to their being loaded into the enclave.

## Critique

Platypus is not the first side-channel attack on SGX, although it is the first one to use power. The vulnerabilities exposed by the Platypus research are not considered to be nearly as serious as Spectre and Meltdown. Mitigating power as a side channel attack for SGX may be as simple as having the processor not update the energy meter when running inside an enclave. Unlike Spectre/Meltdown, however, Platypus is (just) simple enough to be described in some detail to people who are not computer security professionals.

## Code Acts within Network

Returning to the tenets of NMP, we focus here on how Platypus exists within an actor network, which includes a precise, programmable interrupt controller, a fine-grained energy meter, an operating system that provides access to both, and a security enclave that lies beyond the operating system but executes on the same processor cores. Absent any of these, there is either no need for Platypus, or Platypus can't exist. These actants are themselves code, and each comes with its own set of abstractions and vibrancies. Platypus is a study in how these separate code-actants, created in different decades by different teams for different markets, did not quite align well enough to prevent information leakage.

61

Code is not written in isolation. Both exploits and critiques can occur at the joints of the actant-network. At a broader level, Platypus exists because SGX exists, SGX exists because cloud computing exists, and cloud computing exists because of the economics of renting computers to process private information. Part of those economics is the expense of maintaining computers securely in the face of growing and changing complexity across the hardware and system software stack, with that complexity being driven by the perceived needs of the market.

62

## 6. Conclusion

By choosing these particular examples we decentered the text of the code. Asking what Happy Birthday, Platypus, or Firestarter mean could not be answered by a recitation of algorithms used or a summary of code comments. Instead, we had to begin with abstractions, performances, and networks, and from there to paratextual resources: author interviews, peer reviewed publications, and discussions among colleagues. In doing so, we rediscovered two axioms: meaning accrues in the relationship of actants in a network and meaning changes as the network changes. The fact that code abstracts and performs in these fluid networks is what reveals the vibrancy of code.

63

Happy Birthday meant, at the time of its creation, that "computers," such as Mary Ann Mansigh Karlsen, had the freedom to play with a new piece of arithmetic equipment. Decades later its meaning evolved and became, alongside the scientific work she supported, one her most vivid and pleasant memories. In a decade of working with Firestarter, Rountree has seen it transform into a catalyst for revealing unspoken processor design decisions. With Platypus, Daniel Gruss and his students took advantage of the change in the network that occurred when Intel introduced secure enclaves to a platform with sophisticated energy measurement and interrupt control.

64

The contribution of this paper is mapping out the mechanisms and limits of those meanings.

65

**Code abstracts:** Languages allow programmers to work with a simpler world than physical reality.

**Code performs:** Unlike mathematics, code has an embodied physicality of heat, energy, and resonance.

**Code acts within a network:** The network in which the code performs is not fixed, and meaning accrues in the juxtapositions of its actants. If one changes the network, the meaning changes.

**Code is vibrant:** Code cannot be nailed to a single meaning. Meaning is generated within a particular relationship of a particular set of actants, including text, machine, and people. Vibrancy describes what happens as the actants and their relationships change.

Asking what code means is the wrong question. Diversity of meaning is limited only by the diversity and relationships of actants. Instead, ask what the network means.

70

## Works Cited

**Bell 2008** Bell, J. (2008) *American Puppet Modernism: Essays on the Material World in Performance*. New York: Palgrave Macmillan.

**Bennett 2010** Bennett, J. (2010) *Vibrant Matter: A Political Ecology of Things*. Durham: Duke University Press.

- Cancare et al. 2011** Cancare, F., Bhandari, S., Bartolini, D., Carminati, M., and Santambrogio, M. (2011) "A Bird's Eye View of FPGA-based Evolvable Hardware". In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 169-175, doi:10.1109/AHS.2011.5963932.
- Cimpanu 2020** Cimpanu, C. (2020) "New Platypus Attack Can Steal Data from Intel CPUs". *ZDNet*, 10 Nov. 2020, <https://www.zdnet.com/article/new-platypus-attack-can-steal-data-from-intel-cpus/>.
- Colburn & Shute 2007** Colburn, T., and Shute, G. (2007) "Abstraction in Computer Science". *Minds and Machines*, vol. 17, pp. 169–84.
- Condee & Rountree 2020** Condee, W., and Rountree, B. (2020) "Nonmaterial Performance", *TDR: The Drama Review*, vol. 64, no. 4, pp. 147-57.
- Costan & Devadas 2016** Costan, V. and Devadas, S. (2016) "Intel SGX Explained". *Cryptology ePrint Archive*, Report 2016/086, p. 118.
- Hackenberg et al. 2013** Hackenberg, D., Oldenburg, R., Molka, D., and Schöne, R. (2013) "Introducing Firestarter: A Processor Stress Test Utility", *2013 International Green Computing Conference Proceedings*, pp. 1-9, <https://ieeexplore.ieee.org/document/6604507>.
- Inadomi et al. 2015** Inadomi, Y., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., Lowenthal, D., Wada, Y., Fukazawa, K., Ueda, M., Kondo, M., and Miyoshi, I. (2015) "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing", *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1-12, doi:10.1145/2807591.2807638.
- Kocher et al. 2019** Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019) "Spectre Attacks: Exploiting Speculative Execution", *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1- 19, <https://ieeexplore.ieee.org/abstract/document/8835233>.
- Kocher, Jaffe, and Jun 1999** Kocher, P., Jaffe, J., and Jun, B. (1999) "Differential Power Analysis", *Advances in Cryptology – Crypto 99 Proceedings, Lecture Notes in Computer Science Vol. 1666*, M. Wiener, ed., Springer-Verlag.
- Koss 2003** Koss, A. M. (2003) "Programming on the Univac 1: A Woman's Account", *IEEE Annals of the History of Computing*, 25, pp. 48-59.
- Law 2009** Law, J. (2009) "Actor Network Theory and Material Semiotics", In Turner B. (ed.), *The New Blackwell Companion to Social Theory*. Wiley-Blackwell, Chichester, UK, pp. 141–58.
- Lipp et al. 2018** Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018) "Meltdown: Reading Kernel Memory from User Space", *Proceedings of the 27th USENIX Security Symposium*, pp. 973-990, <https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf>.
- Lipp et al. 2021** Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., and Gruss, D. (2021) "PLATYPUS: Software-Based Power Side-Channel Attacks on x86", *2021 IEEE Symposium on Security and Privacy*, 17 pages, <https://platypusattack.com>.
- Marathe et al. 2017** Marathe, A., Zhang, Y., Blanks, G., Kumbhare, N., Abdulla, G., and Rountree, B. (2017) "An Empirical Survey of Performance and Energy Efficiency Variation on Intel Processors", *Proceedings of E2SC'17: Energy Efficient Supercomputing (E2SC'17)*, 9 pages.
- Peterson 2020** Peterson, M. (2020) "New 'Platypus' Attack Can Extract Data from Intel Chips, But Macs Are Mostly Safe". *Appleinsider.com*, November 11, 2020, <https://appleinsider.com/articles/20/11/11/new-platypus-attack-can-extract-data-from-intel-chips-but-macs-are-mostly-safe>.
- Rountree 2020** Rountree, B. (2020) Personal communication with Mary Ann Mansigh Karlsen, April 2020.
- Rountree and Condee 2021** Rountree, B. and Condee, W. (2021) "The Nonmaterial Mirror: Performing Vibrant Abstractions in AI Networks", *Theatre Journal*, 73, pp. 299–318.
- Schechner 2006** Schechner, Richard. (2006) *Performance Studies: an Introduction*. 2nd edition. pp. 1-31.
- Sperry Rand Corporation 1959** Sperry Rand Corporation. (1958, 1959) "Basic Programming: Univac1 Data Automation System", [http://www.bitsavers.org/pdf/univac/univac1/UNIVAC1\\_Programming\\_1959.pdf](http://www.bitsavers.org/pdf/univac/univac1/UNIVAC1_Programming_1959.pdf).
- Winget 1950** Winget, R. (1950) "Locksmiths Used to Worry about Yeggs: Now It's Spies", *Louisville Courier Journal*, April 16, 1950, p. 92, <https://www.newspapers.com/newspage/110607246/>.





This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.