Introduction

Error, Failure, and Code Creation

Ellen Ullman

You need the willingness to fail all the time.

Those words guided me throughout all the years when I worked to become a decent programmer, as they no doubt guided countless others. That one sentence reminded us that coding is a life in which failure will be your constant shadow. Bugs, crashes, halts, glitches, hacks: programmers who want to survive in the profession (like anyone hoping to create a new thing on earth) must come to a begrudging acceptance of failure as a confounding helper, an agent of destruction you wish you could evade, but never can.

The words were spoken by John Backus, who led the group that created the FORTRAN programming language, fully released in 1957.[1] FORTRAN (short for Formula Translator) was the first language that allowed programmers to write code that was not directly tied to any one computing environment. It was a frustrating project that lurched from failure to failure. Backus went on to say:

> You have to generate many ideas and then you have to work very hard only to discover that they don't work. And you keep doing that over and over until you find one that does work.[2]

He also told us:

> If you are not failing a lot, you are probably not being as creative as you could be—you aren't stretching your imagination.[3]

Software companies try to avoid serious failures with procedures, rules, reviews. But programs are works of the imagination that must then make the hazardous crossing into the structured world of code. The attempts to avoid failure will also fail.

All code has flaws, inevitably. Human thought is wonderfully chaotic; it allows us to hold incompatible beliefs, be interrupted, function in a world we do not fully understand. So much of what we know is inscribed in the body, the product of evolution, instinctive, not readily accessible to the rational mind, what Daniel Kahneman has described as fast thinking (System 1). Meanwhile, code-writing (as opposed to the creative work of code-design) requires fully conscious and rational thought, Kahneman's "slow thinking" (System 2),[4] a level of focused attention that is impossible to sustain over time.

I have a friend who was once in charge of testing at a startup that was frantic to go public. The IPO was delayed for months on end because of the relentless appearance of new serious bugs. The higher-ups demanded to know, "When will all the bugs be found?" It was a ridiculous question, because the testing was being done even while new code was being written. Meanwhile, "fixes" to already discovered bugs were in the business of creating a whole new pile of bugs. In any case, no one can predict when the last bugs will be found, because the only correct answer is, "Never."

Many bugs are blind spots in the code. The designer and programmer try to protect the system by looking for conditions that will break things: they will not find them all. Most often, software teams are rushed. They have to create systems quickly. Programmers don't have time to lean back, think of other things, let the background of the mind speak. A prime source of bugs is absurd scheduling.

Other bugs are like physical vulnerabilities inscribed in the DNA. These bugs sit quietly until some environmental factor (in humans, things like ageing, chemicals, medications) suddenly activates the flaw, and we get sick. In the case of computing, the

technical environment is complex and constantly changing. Programs interact with modules not foreseen in the original design; with new operating systems and changed ones, variations in chips, network configurations, protocols, device drivers; bedeviled by documentation that cannot keep up with the changes. What worked one day doesn't work the next, and the programmer's constant question is, "What changed?" Well, lots of things. Which one (or ones) did the damage? That way lies madness.

The deepest weaknesses are revealed when a digital creation is designed for expert users in a collegial environment, and then opened to a wider pool.

Dennis Ritchie and his team developed the C language,[5] which, along with Unix, was part of a research project conducted inside the storied Bell Labs technology incubator.[6] The language gave the team's programmers a great deal of freedom, including direct access to the contents of memory, something systems normally do not allow, in order to protect the integrity of the coding environment. That level of freedom was fine as long as their work remained a research project. According to Brian Kernighan, who coauthored the book that introduced C to the world,[7] Ritchie did not anticipate that the operating system and language would become "as big as they did."[8] Yet they did indeed become big. Programmers' access to memory then escaped into the wild: programs acquired the dangerous ability to invade and manipulate the memory space of another program (mostly by accident), and that invaded program can invade another's (and so on), enabling a world of perplexing bugs.

Then there is the Internet itself, derived from the ARPANET, which was created as a platform in which a limited group of researchers could converse openly about scientific subjects.[9] Security was not assumed to be needed. And so arrived the hackable digital universe.

I once had the good fortune of working for a hacker. This goes back to the time when "hacker" was an honorific, as it still is

among talented hardware and software engineers. It refers to a type of crusty programmer who can chop through code with a combination of grim determination and giddy enthusiasm. The goal, above all, is to uncover the flaws that induce the failures, then (somehow or other) devise the fix that will make things *work*. Their solutions are often "ugly," in coder parlance (aka kludges), the product of down-and-dirty plumbing. But no matter. Maybe lovely, elegant programs and systems can come later. Or not.

"Hacker" has acquired a less admirable meaning, of course, having acquired the taint of what we used to call "crackers," as in safe crackers, people not allowed to get at what's in the safe but who get in anyway. It is a chaotic world involving everyone from cryptocurrency tinkerers to bank thieves; from hackers working for hostile nation states to ones stealing data for espionage and ransom; to those seen as ethical hackers, who want to reveal the wrongdoings of anyone or anything in power; to loners looking for notoriety; to pranksters, jokers, naughty boys of all ages, breaking in just to see if they are clever enough to do it. (It's *fun* to make porn appear in Zoom meetings, isn't it?)

There are the workaday hacks, the constant reports of code vulnerabilities. Peter G. Neumann, the revered computer science researcher, moderates "The Risks Digest,"[10] which is updated weekly, sometimes as often as every few days. The "Crypto-Gram Newsletter,"[11] written by noted security analyst Bruce Schneier, is released monthly. As individual programmers and software makers struggle against the onslaught of flaws in their own code, they are meanwhile bombarded by the hacks that rain down upon the digital planet, nearly invisible, like the solar wind.

Then come the hackers who break into the code meant to defend against hackers: code to protect code becomes a victim. NASA stored reports of vulnerabilities they received from friendly hackers, and then the store *itself* was hacked.[12] Software written by the company CodeCov,[13] which is widely used to test for bugs and code vulnerabilities, was broken into by Russian

hackers, giving them a window into the very code to be protected. In a recently revealed 10-year-old hack, Chinese spies broke into RSA's cryptosystem.[14] The company is a corporate security giant whose customers include "tens of millions of users in government and military agencies, defense contractors, and countless corporations around the world," according to wired.com. The break-in allowed "hackers to instantly bypass [RSA's] security system anywhere in the world."[15]

The fate of humanity hangs in the balance. Nicole Perlroth's book *This Is How They Tell Me the World Ends: The Cyberweapons Arms Race*,[16] describes how the Internet—buggy and hackable—has become a potent military tool. It has the dark power to ignite global war: by accident, or by design.

Now I will return to the "good" use of hacker, because I want to preserve its historical meaning among the general public and give the original hackers their due: an army of sometimes disheveled geniuses who were wary of rules and formalities, nonconformist in their thinking, somehow both brilliant and practical at the same time, who could reach in, rummage around, and figure out what to do. A member of Backus's FORTRAN team called their group "the hackers of those days."[17]

A now-famous hack saved the Apollo 13 mission from disaster.[18] Before the mission could achieve a moon landing as planned, an oxygen tank exploded in the command module. The three astronauts had to take refuge in the lunar module, which was designed to carry only two astronauts. To reduce the build-up of carbon dioxide, they retrieved an additional canister of lithium hydroxide pellets (a carbon dioxide scrubber) from the command module.[19] But there arose the sort of problem that plagues complex projects: components designed and built separately. One canister had a round connector, the other a square one, the proverbial square peg in a round hole.[20] A remedy had to be found—quickly—or all three men would die of asphyxiation.

NASA engineers on the ground raced to find a solution. They threw together bits of stuff that were on the spacecraft—plastic bags, covers ripped from manuals, duct tape, cardboard, anything—and devised a bridge between the mismatched connectors. It was one of those "ugly" fixes. As the Apollo 13 astronaut James Lovell later described it: "Hose. Duct tape and an old sock."[21]

Daniel Kaminsky, a famed cybersecurity expert, created another legendary, down-and-dirty hack. In 2008,[22] he discovered a security hole in the Internet's Domain Name System (DNS), which converts website URLs to specific IP addresses. Kaminsky saw how easy it was for knowledgeable bad actors to redirect the user not to the intended destination, but to a world of fake sites—a "bank," a "credit card company," an "email login"—and therefore collect the user's IDs and passwords. He alerted others and, along with Paul Vixie, coded an emergency patch.

Kaminsky, who will forever have a place of honor among the greats of the hacker community, died on April 23, 2021. His obituary in the *New York Times* called him "the savior of the Internet."[23] He was the first to sound the alarm and respond to the threat. Yet, given what we know about the relationship between coding and error, it is no surprise to learn that the patch was far from perfect. After the "fix" was installed, there were 6:00 a.m. calls from Finnish certificate authorities saying their security procedures were broken. Some DNS servers stopped working correctly. And there were some pretty harsh words from Kaminsky's peers in the security community.[24] Years later, in a talk at the 2016 Black Hat hacker conference, Kaminsky referred to his patch as "that DNS mess."[25] Vixie, a longtime steward of the DNS, described the code they cobbled together in terms yet more ugly than Apollo's old sock: he compared it to dog excrement. In the way of hacker expediency, he called it the best dog excrement "we could have come up with."[26]

Each of the programs, systems, and concepts discussed in this book had to go through the test of trial-by-error. The essays in

this book explore a wide range of topics. Several offer a deeper look at technologies familiar to the general public: the coming of Email, hyperlinking, JPEG image files, the Facebook Like. Some discuss historical landmarks that ought to be known more widely: women's contributions to early computing; the creation and endurance of COBOL, the first language in general use for business software; the coming of BASIC, the wonderful beginner's language.

Two essays explore deeper concepts in computing: data encryption, and the Markov Chain Monte-Carlo concept (MCMC), a foundational mathematical method used to understand distributions in data and arrive at probabilities.

Computing can bring happiness, as three essays show. There is pleasure (and pain) in learning to write code; in the fun brought into the world by Spacewar!, the first distributed video game; and in the advent of the Roomba, which, in addition to cleaning floors, also gave hours of delirious pleasure to innumerable cats.

Two essays discuss contributions to computing that I see as being derived from the idea of "the wisdom of the crowd": the Facebook Like button and page ranking. The premise is that numbers in and of themselves say something about the worth of whatever is being liked, from websites to Instagram postings to dance crazes on TikTok: more likes equals more eyeballs equals "better." The underlying theory is based on the belief that, given a very large universe of participants, a truth will emerge.

The coming of the "smart mob" has been a decidedly mixed blessing. Twenty-five years ago, I had an informal talk with Larry Page about Google's search engine as it worked at the time. I said I was concerned that the order in which results were listed, based as it was on the number of links into a given page, was a species of the rich getting richer. Larry, ever thoughtful, sat quietly, considering his reply. Finally he said, "I worried about that too, but I realized there was nothing I could do about it."

What he meant was that there was nothing he could do *algorithmically*. Given the immense universe of knowledge, a human

curator would have faced an impossible task; code has to be the curator. Google's search engine has improved vastly over time, its criteria for ranking becoming ever more sophisticated. And search engines, most modeled on Google's, have brought astounding advances in how human beings can understand the world. Yet search engines have also ushered in the age of "most popular," "trending," "bests," and posts that users hope will "go viral." This amplification of responses can empower the public and create a world of fun. They also reveal the hazards of assigning wisdom to the crowd: results prejudiced by the cultural majority, an arms race between the search algorithm and sites wanting to promote themselves, conspiracy theories, hordes of influencers stoking likes and clicks, truly fake news.

Then there are the programs we wish had not survived the assault by bugs. One essay examines so-called predictive policing, which pretends to predict where crime will take place in the future. Like all AI algorithms, it is based on databases laced with bad information, on methods that are rife with bias.

On a lighter note, there is another maybe-we-never-wished-for code invention: the pop-up ad. The essay here, by the programmer who authored it, describes his remorse, the regret he feels about loosing the pop-up upon the world.

A book about code must necessarily address the subjects that are integral to the creation of software: error and failure. "The Lost Mars Climate Orbiter" describes a failure that, 28 years after Apollo 13,[27] echoes the earlier mission's mistake: system parts created separately. One team used the American measurement system, the other the English Imperial system. The repetition of this type of error shows how pervasive are the hazards in complex systems, where one group of engineers cannot possibly create the whole, and disparate parts must somehow be knit together, and flawlessly.

"Heartbleed" describes a bug deep in the internals of the Internet that caused havoc for millions of devices. A hacker ex-

ploited weaknesses in open-source software and vulnerabilities in the C language, as mentioned above, which gave programmers direct access to the contents of memory. Like so many errors, the problem lay dormant, everything apparently working, until something in the environment changed: the arrival of a hacker with malicious intent.

Another essay discusses the Morris Worm, the first to be distributed via the Internet. Robert Tappan Morris, then a graduate student at Cornell, wrote the invasive code as an intellectual project, as a test of the Internet's weaknesses. However, a mistake in his code instructed the worm to keep reproducing itself, whether or not a system had already been infected. Then he inadvertently released the worm into the wild. A senior engineer who worked on the emergency caused by the worm, Colm MacCárthaigh, later said, "It felt like the Internet was on fire." Morris never intended to cause the vast damage he did. In this sense, his worm was a bug inside a hack.

A particularly pernicious use of errant code was deployed by Volkswagen to falsely lower the readings of pollution levels caused by their diesel engines: an intentional bug, an error created for corporate gain.

And then we come to the day-to-day, unglamorous but vital chore performed by all good programmers: adding comments to their code. Comments are an invaluable tool; they describe sections of the program that are tricky, not immediately obvious or readable. Comments are acts of generosity, help for the unknown colleagues who will work on the code over time, in the hope that they will keep a system working.

Sometimes the "future" programmer will be the original author of the code, and the comment is a gift to oneself, since it is all but impossible for individuals to recall all the complex details in the software they have written. A bug is an opportunist that waits at the gate of any change to the body of running code; a comment is a weapon that, *a priori*, takes up the battle against software entropy.

I am just old enough to remember the desperate attempts by the United States to match the Soviet Union's great achievement, Sputnik, the first earth-orbiting satellite. NASA's launches were broadcast on television, some live. We saw one rocket after another exploding spectacularly on the pad; or collapsing in a ball of fire after lifting-off a mere few feet; or managing to rise into the sky only to burst into flames at the first stage of separation.[28] Those failures are engraved in the memories of those who watched the attempts: the great anguish inherent in technological achievement, and, per Backus, the imperative to try again.

Decades later, after scores of intervening successes—including a human's trip to the moon and projects that sent explorer satellites to the edge of our solar system and beyond—NASA launched the mission to send the Perseverance Rover to Mars. The launch took place on July 30, 2020.[29] On February 18, 2021, nearly six months later, Perseverance landed on Mars.

The landing was streamed live[30] thanks to NASA's commitment to inform the public, even if a mission might fail. What riveted my attention was a pane on the left side of the screen. It highlighted each stage as the mission unfolded, modules for launch, separations, cruise balance, etc. Between each module was a step that began with the word "`Interface`," as in: `Module A, Interface to module B`, `Module B, Interface to Module C`, `Module C`, and so on. You could see the tension in the faces of the women and men staring into their monitoring screens. I held my breath along with them.

There is no more hazardous place in a complex project than the handshake between one section and the next. In this interregnum lurks all the potential misunderstandings between separate groups of developers, as we saw with the lost Mars orbiter and the near catastrophe of Apollo 13. The illuminated word "`Interface`" always seemed to linger for far too long. I wondered if this latest generation had learned the lessons of their forebears, who knew the danger zones. In the case of a

breakdown, did these young engineers have the hackers' skills to scrounge around and repair a ripped seam? This Mars Rover project seemed impossibly complicated, riddled with opportunities for disaster. I watched in a mood of both exaltation and horror.

Time went by. The display followed the steps in the project: one module, interface, next module, interface, and the next. Finally we came to the astounding unfurling of the parachute that gently lowered Perseverance to the surface. And it was done.

And yet.

There is no such thing as the last bug.

The problem appeared in the initial test of the small helicopter, Ingenuity, which had arrived on Mars attached to the underbelly of Perseverance, like a baby kangaroo in the pouch of the mother ship. Ingenuity was to attempt to fly in the thin atmosphere of Mars, to pioneer an age of powered, controlled flight—engineered by humans—on a planet other than earth.

The first try failed. The helicopter's start-up activities took longer than expected, and its computer shut down the motors. The engineers overseeing the mission identified a potential workaround and devised a patch. Yet, knowing that touching existing code is an excellent opportunity to break it, they wisely did not install it. Instead, they adjusted the commands they would send to the craft.[31]

Here was a repair that was sent not through the Internet but across 130 million miles of space.[32] Engineers had to wait two anxious earth days to find out if their changes would work.[33] On April 19, 2021, Ingenuity rose 10 feet into the Martian atmosphere as planned, hovered briefly, banked, turned, and landed at its takeoff point.[34]

More flights followed. Failure had led to success. This was a bug-fix for our time, another hack for the ages.