THE NEW ATLANTIS
A JOURNAL OF TECHNOLOGY & SOCIETY

Buggy Software and Missile Defense

Mark Halpern

Since the days when the proposed Anti-Ballistic Missile (ABM) system was derisively called "Star Wars," one of the principal arguments against it has been that it would require a degree of software perfection that is simply not achievable. Some of the world's preeminent computer scientists—like David L. Parnas, now of the University of Limerick—have been saying this for years, and their views have been widely publicized in both the technical and popular press. Their argument is straightforward and even plausible: a software system of the necessary size and complexity for missile defense would be paralyzed by bugs, impossible to test thoroughly, and certain to be overwhelmed by the complications and distractions that an attacker could easily throw at it. Opponents like to characterize missile defense as "trying to hit a bullet with a bullet," as if doing so is obviously impossible.

Of course, these software arguments against the ABM project are not the only ones its opponents mount against it; there are a number of other technical issues, not to mention various political and economic ones. But the software problem is serious enough, visible enough, and interesting enough to address here—both to correct our reigning confusions about software in general and to demonstrate that the software case against missile defense is ultimately flawed.

Is Software Inherently Buggy?

I am hardly blind to the problems of buggy software. I have been programming and designing software for close to fifty years, and the practice and theory of debugging has occupied a great deal of that time. I started writing about the problem of buggy code as long ago as 1965, and one of my dicta in an early paper attained some currency: "That tendency to err that programmers have been noticed to share with other human beings has often been treated as if it were an awkwardness attendant upon programming's adolescence, which like acne would disappear with

---

Mark Halpern *has been working in and with computer software for fifty years, starting out with IBM's Programming Research Department just after the release of Fortran, and going on to work for several other companies, including Lockheed Missiles & Space Company, tiny Silicon Valley startups, and then IBM again. He lives in the hills of Oakland, California, with his wife and daughter. His website is www.rules-of-the-game.com.*

the craft's coming of age. It has proved otherwise." And like anyone who uses computers daily, I have more stories than I'd like to recount about software crashes at inopportune moments. So: If I reject the view that software problems make ABM impossible, it is not because I naïvely think that software problems in general are not real.

Everyone knows that almost all commercially marketed software is buggy, certainly when first released, and often even after years of use. We have all been told that we cannot find out whether a certain check has cleared because "the computer is down," or that our motel reservation was lost because of "a computer glitch," and we know this usually means that the software has failed. Such experiences have turned us all into computer cynics, rolling our eyes at claims by software manufacturers emphasizing their products' reliability and freedom from crashes. And many of us—including some professors of computer science—have concluded that software bugginess is an inherently intractable problem, and that it will remain so until some great intellectual breakthrough occurs.

This conclusion is wrong. Computer software, far from being intractable, is the most docile, well-behaved tool ever developed. It can be tested nondestructively, it can be repaired cheaply and on a piecemeal basis, it never wears out, and—most importantly—it responds to identical inputs under identical conditions with absolutely identical outputs, forever. No hardware artifact offers behavior so totally consistent and so amenable to testing and repair. And none, for just those reasons, offers more potential reliability and perfectibility.

Some critics, like Parnas, have argued that software lacks *dependability*. In a 1990 article, for example, he and colleagues claim that "within the engineering community software systems have a reputation for being undependable." True, software systems have a reputation for taking three times longer to complete than predicted at their outset, and costing more by an even higher factor. But undependability is not part of software's Original Sin. Once some real-world experience is acquired in the use of such a system, it is usually one of the most dependable components of any engineering project. Those experienced in such matters, who understand that dependability means not "ability to guess what I want" but "consistency in reacting to given inputs," will have no trouble in deciding between Parnas's view and my own.

Of course, the malleability of software constantly tempts us to revise it, and we succumb to temptation frequently, with the result that the software instantly and without warning reverts to pre-release condition, with all the shakiness that implies. And of course, those who write software for the consumer market often yield to the inevitable pressure to release a

software product before it is really ready to be put into customers' hands. But it is unreasonable to call software undependable when what we mean is that we are undisciplined. *We* are the enemy; we take advantage of software's amenability to modification, as well as the unlikelihood that customers will discover the bugs in certain less-frequented paths through our code, at least before the warranty period is over. That we can behave as badly as we do, and still turn out products that mostly work, is a tribute to software's outstanding dependability.

In addition, confusion about software is caused by the completely subjective way in which we use terms like *expensive* or *complex*—a way that makes them simply indicators of our feelings at the moment, rather than saying something useful about the objects we're contemplating. If we say that a piece of software is expensive when its function could not be realized in any other way, what metric are we tacitly appealing to? We must be comparing the cost of the software to *something*, since it makes no sense to call anything expensive absolutely. Consider the following thought experiment: Can you give a simple yes-or-no answer to the question, "Is $1,000 a lot of money?" Would your answer be the same regardless of whether the question continued "for a beachfront house in Malibu" or "for a peanut butter sandwich" or "for Aladdin's Lamp"? In much the same way, absolute characterizations of software as expensive are nonsense; what they usually signify is our disappointment with our own performance as project managers and programmers, and our need to find someone or something to blame.

We also speak subjectively of complexity. When we first learn how to drive, for example, we are overwhelmed by the apparent complexity of a car's controls, let alone the bewildering chaos of traffic; after a few years of driving, we step into a rental car of a make and model we've never handled before and drive away in thirty seconds without hesitation. Similarly, the computer seems at first formidably complex; later it will seem so simple—and simple-minded—that we will have to remind ourselves not to kick it.

Even experts who should know better speak loosely of complexity in the context of software. In the article mentioned above, Parnas and his coauthors say, "The most immediately obvious difference between software and hardware technologies is their complexity"—meaning, as the context makes clear, that software is much more complex than hardware. They are wrong. The programmable computer is, in an important respect, the *simplest* machine man has ever invented: it is the simplest way of achieving virtually any functionality that can be expressed as an algorithm. That simplicity may not be evident to inspection or intuition, but quickly becomes evident by comparing its cost-per-function with that of any possible alternative.

For the class of functions that we program computers to perform—such as word processing, massive computation, searching through huge databases, and the generation of graphics—the software that performs those tasks is incomparably easier to create, more testable, and more reliable than the equivalent hardware (if such hardware could even exist).

To be sure, we often build software objects that are very complex and troublesome—not compared with some other way of realizing their functionality, but by the standard of what we humans can control and remember. This is a consequence not of software's supposed intrinsic complexity, but of human insistence on pushing everything to the limit. That we often get into trouble using so simple a technology is not a reflection on the technology, but a reminder of man's permanent role as troublemaker. Given any new tool or technique, we extend it until we make a mess of things; that's been our way since we ate the apple, and we'll be doing it when we storm heaven. Calling software complex is rather like calling bricks complex because, using them, we can construct bizarre and bewildering labyrinths.

In the end, the only rational basis for measuring the relative complexity of software and hardware is *function-by-function*—that is, by comparing the costs of realizing well-defined functions using each of the competing technologies. Since this is an exercise we are unlikely to carry out in reality, we can only run a thought experiment—but that is all that should be necessary. Microsoft Word occasionally crashes or does strange, inexplicable, undesirable things. But consider what kind of machine we would have to build and maintain to realize Word's functionality in hardware. Most likely, that application would not exist at all, because the necessary hardware would be too difficult and too expensive to build. A great many comparisons between software and hardware are invalidated by the same consideration.

The Demand for Buggy Software

But if software is as docile and simple as I contend, why are programs so buggy? A general answer has already been given: because it is human nature to push until we get into trouble—and then blame our tools. We load the elephant with feathers until the elephant collapses, whereupon we conclude that feathers are too heavy for elephants. No matter how amenable software is to our efforts, it can overwhelm us if we pile the code high enough—and we often do, because it's so fatally easy. But the special reason for software's bugginess is that we almost never *demand* that it be bug-free (I use "demand" here in the economist's sense: not just desire, but desire backed up by ability and readiness to pay).

Software manufacturers are rational economic actors; if they can sell us software without going to the expense of thoroughly debugging it, they will. The copy of Microsoft Word that occasionally drives me crazy cost around $200; if Microsoft had been forced to debug it thoroughly before releasing it, its price would be closer to $2,000. Would I pay that much for a version that I could be sure would never crash at a critical moment, losing hours or days of my work? Probably not; apparently, I don't value my sanity that highly. I am neither blaming anyone nor apologizing for anything; I am simply reporting Microsoft's behavior and mine, in the belief that they are typical of just about all software developers and computer users. In a word, we have buggy software because we consumers won't pay what effectively bug-free software would cost.

The reasons why software is almost always buggy are not inherent in the technology and thus inevitable, but spring from human choices and practices that we can understand and could change if there were a compelling reason to do so. Those habits include piling the code on until it overwhelms us, and taking our chances with buggy software in order to get it more cheaply. Both problems could be overcome if we wanted to overcome them badly enough. And in building an ABM system designed to guard us against missiles carrying nuclear warheads, we would surely want to.

In asserting that the reasons for the prevalence of buggy software lie in human habits and choices rather than in the nature of software, I find support from an unexpected quarter: Parnas himself. In an interview given in 1999, he related this anecdote about his experience as a consultant to industry:

> **Interviewer:** What are the most exciting/promising software engineering ideas or techniques on the horizon?
>
> **Parnas:** I don't think that the most promising ideas are on the horizon. They are already here and have been here for years but are not being used properly. A few years ago, I met an experienced software development manager who had just uncovered a memo I wrote for his company in 1969. He told me, "If we were now doing what you told us then, we would be far ahead of where we are now." The biggest payoff will not come from new research but from putting old ideas into practice and teaching people how to apply them properly.

His story is one that many consultants to the computer industry could echo: again and again, they have shown clients how they could turn out

far better, more reliable products, and again and again those clients, while conceding the technical merit of the recommendations they have paid so much for, have failed to implement them. I emphasize again that this is not a criticism of either the consultants or of the companies that seek and then disregard their advice. The managers of the companies involved have many factors to weigh in deciding whether to implement such proposals, and those who are unfamiliar with the particulars of a given company's situation should not presume to fault them for their decisions. Maybe the company alluded to by Parnas would have gone bankrupt if it had tried to implement, then and there, the recommendations he made in 1969. The larger point is that the path to better software, even effectively bug-free software, has been clearly marked for a long time; what has been lacking is not some technical breakthrough, but the will and the means to take that path.

You Get What You Pay For

And this brings us to the specific question of missile defense. If I am correct, there is nothing intrinsically problematic about building the necessary software, as ABM opponents like Parnas imply. We can have software that is virtually perfect if we are willing to pay for it, and the U.S. government surely has the resources to do so for so important an objective. If software is created not for sale but for national survival, and if its development is supported by resources commensurate with that objective, there is no reason why it should not be virtually bug-free and thus the least troublesome part of an ABM system. The project would require a large initial investment, and it would take far longer to produce its first line of code than the conventional method of producing software for a mass market, beginning with a long period of constructing tools and test-beds. Yet precisely because it enjoyed the use of those tools and tests, it might very well *finish* no later than a conventional software project of comparable size. And it might even prove to be a money-saver in the long run, once we consider all the expensive problems that the user code so produced would *not* exhibit during its lifetime, and once we take into account the uses for all that initial tooling and test-building on other projects.

This is not, incidentally, just a plea for more care or "good practices" in software development; I have in mind very specific, concrete steps to achieve thoroughly debugged software products. There is a long history of serious studies and proposals on how to build really robust software. But like the one Parnas made as a corporate consultant, they were never

implemented because there was no demand for robust software at the price it would have cost. These studies began appearing as long ago as Douglas McIlroy's proposal in 1968 for the development of reusable software modules, followed by the proposal by the late Robert W. Bemer (one-time Director of Software Standards at IBM) for a Software Factory; they are still being refined today, as in my own Assertive Debugging System proposal, recently described in a leading programming journal ("Assertive Debugging: Correcting Software As If We Meant It," *Embedded Systems Programming*, June 2005).

Profit-making companies, worried about how Wall Street will view this quarter's results, may seldom if ever find a business case for making the initial investment necessary to move to such systems or methodologies; the richest nation on Earth, trying to protect itself from a terrible disaster, might well employ other criteria than that of short-term return on investment in deciding how much it should invest—just as it did, for example, when it created the Manhattan Project. But is missile defense a sensible response to the threats and challenges of today's geopolitical world? And what impediments, beyond software troubles, still stand in the way? I have neither the expertise nor the space to give full and adequate answers to these questions. But I can at least try to correct some grave technical misconceptions.

Can We Hit a Bullet with a Bullet?

Opponents of the ABM project frequently claim that any enemy technically advanced enough to mount a missile attack would also be advanced enough to incorporate many kinds of dummy warheads and decoys into that attack, forcing us to try to shoot down an impossible number of objects in an attempt to destroy all the real warheads. And even though the tests run so far have shown more than once that an ABM *can* intercept a target missile following a trajectory of the kind that would be taken by an actual intercontinental missile, we are still told that the ABM system, in attempting the supposedly quixotic feat of "hitting a bullet with a bullet," is doomed to failure. To this claim there are several counter-arguments:

1. Most of the ABM tests to date have been conducted under so many arbitrary constraints, such as the technical handicaps imposed by treaties and other political considerations, that the wonder is that they have been as successful as they have. The kind of ABM system that the United States will be able to deploy, freed of such constraints and allowed to use all its technological muscle, should be far better than the crippled systems

displayed so far. That even the rudimentary system so far tested has been able several times to destroy its target is a most promising indication.

2. It is by no means obvious that a nation or group possessing a few ballistic missiles would also have dummies and decoys. These deceptive devices are far from simple to design, construct, test, and deploy; those that try to mimic warheads may have to be in some ways more sophisticated than the warheads themselves. They are also expensive. Insofar as the proposed ABM system is simply a defense against "rogue states" such as North Korea or Iran, it should be able to do its job for some years to come without worrying about advanced forms of deception.

3. "Hitting a bullet with a bullet" is good sloganeering, but nothing more. Its effectiveness lies in its ability to plant a false picture in our minds: it suggests a human marksman attempting an utterly superhuman feat. But the feat of hitting a bullet with a bullet—already successfully done several times, with far from optimum systems—is in fact not particularly difficult for a computer-based system, which in this respect *is* superhuman. A bullet or other purely ballistic missile is one that follows a predetermined course, a course that can be predicted with great accuracy from just a few observations. And the bullet that is fired at that bullet—the ABM—is another such purely ballistic missile until its final moments, when it is freed to make whatever last-minute adjustments in its course may be necessary if it is to come within lethal distance of its target. This is exactly the kind of work that computers are good at: predicting the course that an enemy missile will take, and determining the course that an ABM must follow in order to intercept it. In fact, every time we launch an artificial satellite into its prescribed orbit, or send astronauts up to the International Space Station, or send a probe to explore the rings of Saturn or hit a comet, we are doing something comparable to "hitting a bullet with a bullet"—and we succeed so routinely that our newspapers put the stories back on page 27.

4. The kind of computation ABM computers will have to do is well understood. They will be called on to process inputs from sensors such as radar and infrared detectors, deriving from those data the positions and momenta of objects either in powered trajectories or in free fall. This processing will predict where those objects will be at later times, and where other such objects—or beams of energy or particles—must be directed in order to intercept them. This is a kind of processing in which we have a great deal of experience, and which requires no scientific breakthroughs or even much new software. There are certainly some stringent requirements that ABM computers must meet, but they do not involve

the generation of much new code. The first requirement is *speed*—these computations will have to be performed faster, probably several orders of magnitude faster, than the relatively leisurely computations required by a mission to the Moon, for example. The second requirement is *radiation resistance*. Insofar as the computers and other electronics are located outside the Earth's atmosphere, they will be subject to ionizing radiation that would be destructive to devices built of ordinary, home computer-grade components. These two requirements are imposed on the hardware—a technology where we are making rapid progress, and may have made enough progress already—and not on the software, where masses of fresh code, if conventionally generated, would mean a swarm of new bugs.

5. The bullet-hitting-a-bullet argument is beside the point in another way, too. ABM research and development is being devoted not only to hit-to-kill weapons that would have to intercept an enemy missile or warhead by hitting it directly, and when in full stride, but also to ABM weapons that would not have to meet even those requirements. These alternative ABM weapons would attack enemy missiles when they are much more vulnerable (i.e., during their lift-off phase, when they would be moving relatively slowly, and still over enemy territory); with a laser beam, which flies straight and at the speed of light; or with weapons that do not distinguish between warheads and decoys but simply detonate with enough force to destroy any targets, real or decoy, within a sphere of such large radius that pinpoint accuracy would be unnecessary.

In the end, the technical problems posed by an ABM system are of the kind that the United States, in particular, has always been good at handling. Large-scale, highly ambitious engineering projects are our specialty—think of the Panama Canal, the transcontinental railroad, the Manhattan Project, the Polaris Submarine-Launched Missile project, the Apollo lunar landing, the International Space Station, the Hubble Space Telescope. What's more, we have a history of success with military or space engineering projects that involve huge amounts of programming—such as the space shuttle, the Cassini probe of Saturn, and various manned and unmanned military aircraft. Why should we believe that the country that succeeded in all of these will not succeed also in the development of an ABM system, if it decides to do so?

A Time to Act

A technology like missile defense must take its direction from policymakers, who must grapple with defining the nature of current threats and discern-

ing the wisest use of resources to meet those threats. During the Cold War, the point of an ABM system was largely deterrence; its "success" depended less on its capacity to shoot down missiles and more on its capacity to convince the Soviet Union that it could shoot down missiles. Today, our threats are perhaps less well-defined, and thus the political and strategic purpose of missile defense is perhaps more multifaceted. We want a system that deters enemy states from an attack, a system that deters rogue actors from investing huge sums of money on missiles that may prove futile, and a system that actually does what it is programmed to do, if and when a non-deterrable actor attacks us.

It is commonly urged that there are many dimensions to the ABM project and many consequences that cannot now be foreseen. But these caveats and warnings apply to *any* possible handling of the security threats that the ABM proposal is meant to deal with—including doing nothing, relying on diplomacy and treaties (conceived as an alternative to weaponry, offensive or defensive), promoting democracy abroad, or disarming unilaterally. We have to do *something*, and that something must be determined by the best information we have at our disposal. I submit that the best information we have today supports proceeding aggressively with missile defense:

1. There is every reason to expect the technology will work. The ABM project is of exactly the sort that we have succeeded with in the past. No major scientific breakthroughs are required, and no large amount of new software need be written.

2. The ABM system's reliance on software is not a weakness but a strength. Software, unlike hardware, is perfectly stable; once debugged, it will remain in that state long after every hardware component of the system has rusted or otherwise deteriorated. And getting it debugged—which means not the logically impossible task of proving it free of defects, but building it with tooling that precludes whole classes of common bugs at the outset, and testing it as thoroughly as human ingenuity and persistence is capable of—is just a matter of devoting sufficient resources to the task.

3. It is true that a missile defense system cannot be tested in a totally realistic way—which is to say, we aren't going to provoke a hostile country into launching an attack on us just to see if our defense works. In this respect, missile defense is just like all other proposals for defending ourselves against attack from weapons of mass destruction. But such "perfect" testing is not necessary. By creating the necessary software with the help of the advanced development methods mentioned earlier, and applying the

most exhaustive testing our resources permit, we can develop an effective missile defense system—one with a likelihood of success in action sufficient to deter any rational enemy, and with a reasonable chance of providing protection in the event of attack by an irrational one. That is all that any defensive system can be asked to do.

Without question, software is not the only technical barrier to an ABM system. But only by setting the software objections aside, once and for all, can we sharpen our critical gaze on the real challenges before us. As a technological matter, we may now have the power to protect our country from the kind of attack that several of our enemies have already threatened; as a political matter, failing to do so would be a mistake of tragic proportions.