Selected research activities

Daniel J. Bernstein
djb@cr.yp.to

2005.01.07

1 Fast discrete Fourier transforms (FFTs)

My djbfft software at http://cr.yp.to/djbfft.html computes power-of-2 discrete Fourier transforms at extremely high speeds. It's several times faster than a typical "optimized" FFT library. It's even faster than the Frigo-Johnson "Fastest Fourier Transform in the West."

FFTs have a tremendous range of applications. For example, several DVD-playing programs (such as ogle, my favorite) now use djbfft to save time in decompressing movies.

Haven't FFTs been thoroughly understood for decades? From the perspective of algebraic complexity, the answer is yes. The split-radix FFT computes a size-2^k DFT using $(4k-6)2^k+8$ floating-point operations (about 1.25 times faster than the simplest FFT method); this is conjectured to be optimal. I've found improvements in the algebraic complexity of more complicated functions—see subsequent sections of this document—but I'll be amazed if anyone finds a way to compute a size-2^k DFT with fewer than $(4k-6)2^k+8$ floating-point operations.

Real computers, on the other hand, have communication costs not reflected by algebraic complexity. Real computers have a load latency, typically two or three clock cycles; a floating-point-operation latency, typically three or four clock cycles; and several other bottlenecks. (All floating-point operations are carried out inside "registers," a very small amount of high-speed memory; "loads" copy data from slower memory to registers.) Minimizing time on real computers is more complicated than minimizing the number of floating-point operations.

I started my FFT project by identifying the ring homomorphisms behind every known FFT method, and thus condensing the FFT literature down to a few pages. See Sections 7–8 of http://cr.yp.to/papers.html#m3 and Section 1 of http://cr.yp.to/papers.html#multapps. This structure made it easy for me to spot a new FFT method, which I call "exponent -1 split-radix," that reduces the communication complexity of the FFT: specifically, it halves the number of loads of precomputed FFT constants.

(For comparison, Van Loan's book *Computational frameworks for the fast Fourier transform* spends hundreds of pages describing a smaller set of FFT methods. Van Loan uses merely the vector-space structure of the underlying rings, whereas I also take advantage of the structure of these rings as polynomial algebras, drastically simplifying the presentation of the algorithms. I've seen similar effects in many other areas: finding the right mathematical framework

is tremendously helpful in understanding the existing algorithms and in finding better algorithms.)

I then spent time on "instruction scheduling" and "register assignment" for each target CPU: choosing an order of loads and floating-point operations to minimize overall latency. Most FFT libraries trust compilers to do this, but the unfortunate reality is that compilers do an awful job, often losing a factor of 2 in speed.

Later I started working on `floatasm`, a set of tools to help me schedule (and verify) instructions for FFTs and for many other computations. I should emphasize here that there's a fundamental difference between the `floatasm` project and typical optimizing-compiler projects described in the literature. The goal of a compiler is to make a program faster while receiving no help from the programmer; in contrast, I insist on ending up with top speed, and I then ask how the programmer's time can be minimized. Programs are not static objects; the programmer *cooperates* with `floatasm`. The resulting code is faster than the computer would have produced without the programmer's help. The programmer spends less time than he would have spent without `floatasm`'s help.

2 Integer multiplication

My `Zmult` software at `http://cr.yp.to/zmult.html` multiplies two integers modulo $2^{393216}+1$ in $9.1 \cdot 10^6$ Athlon cycles. For comparison, Granlund's GMP library at `http://swox.com/gmp` takes $13.9 \cdot 10^6$ Athlon cycles to multiply two 196608-bit integers, and $40.7 \cdot 10^6$ to multiply two 393216-bit integers.

The current version of `Zmult` is merely proof-of-concept software, supporting only a few input sizes, but I'm turning `Zmult` into general-purpose software that supports arbitrary input sizes—including sizes that don't even fit into RAM! I'm providing a very simple multiplication interface so that GMP will be able to switch to using `Zmult`; every program that uses GMP will then benefit from `Zmult`'s speed.

Large-integer multiplication has many applications—including some recent surprising applications to problems that do not appear to involve large integers. One of these surprising applications is to smoothness detection in the number-field sieve, as discussed in subsequent sections of this document; this application is my main motivation for handling integers that don't fit into RAM.

Underlying techniques

In 1971—see `http://cr.yp.to/bib/entries.html#1971/schoenhage-mult`—Schönhage and Strassen proved that two n-bit integers can be multiplied in time $n(\lg n)^{1+o(1)}$ on a multitape Turing machine.

Most software uses simpler FFT-based methods that take time $n(\lg n)^{2+o(1)}$. The Schönhage-Strassen method, despite its asymptotic advantage, is perceived as being slower: it is constantly multiplying integers by 2^1, 2^2, 2^3, etc., requiring expensive shifts in its inner loop.

The most important point in `Zmult` is that almost all of the shifts in the Schönhage-Strassen method can be eliminated. One can align almost all roots of 1 in the Schönhage-Strassen method to be powers of (e.g.) 2^{32}, in the same way that the split-radix complex FFT aligns most roots of 1 to be powers of $\sqrt{-1}$. The savings for the Schönhage-Strassen method is much larger than the savings for the complex FFT, because there are many more aligned roots of 1.

There's much more to say about the speed of integer multiplication. I've noticed, for example, that some Fermat numbers factor more nicely than average, so a few extra roots of 1 are available to the Schönhage-Strassen method. I've also found a way to reduce the communication costs in the Schönhage-Strassen method below what's possible for simpler FFT-based methods, with a data flow that violates the underlying ring structure. This improvement will be particularly helpful for inputs that don't fit into RAM.

Price-performance ratio for circuits

For obvious reasons, today's popular CPUs include fast circuits (often buried inside the floating-point unit, but sometimes more easily accessible) to multiply small integers, usually 64-bit integers. These circuits account for a significant part of the cost of a CPU.

I've noticed that CPU designers are building multiplication circuits using the same techniques that were used in the 1960s. More recent techniques are regularly used by (e.g.) people writing software to multiply 1024-bit integers, but are ignored by (e.g.) people designing circuits to multiply 64-bit integers.

I'm starting to explore this area. I believe that state-of-the-art multiplication techniques, when carefully applied, will substantially reduce the cost of building a 64-bit multiplication circuit.

3 Factorization into coprimes

One can factor a set of positive integers into *coprimes* in polynomial time. The input is a finite set (or multiset) S; the output is a finite coprime set P together with the factorization of each element of S as a product of powers of elements of P.

The obvious algorithm—look for $a, b \in S$ whose greatest common divisor $g = \gcd\{a, b\}$ is larger than 1; replace a, b with $a/g, g, b/g$; repeat—takes at most cubic time.

Bach, Driscoll, and Shallit introduced an algorithm taking at most quadratic time. See `http://cr.yp.to/bib/entries.html#1993/bach-cba`.

I found an algorithm that takes essentially linear time: more precisely, time $n(\lg n)^{O(1)}$ where n is the number of input bits. Recently I found an algorithm that takes time at most $n(\lg n)^{4+o(1)}$. See `http://cr.yp.to/papers.html#dcba` and `http://cr.yp.to/papers.html#dcba2`.

My algorithms start from the Schönhage-Strassen algorithm to multiply in time $n(\lg n)^{1+o(1)}$ and the Lehmer-Knuth-Schönhage algorithm to compute gcd

in time $n(\lg n)^{2+o(1)}$—see http://cr.yp.to/papers.html#multapps, Section 22. My algorithms then add several additional ideas.

Applications

Factorization into coprimes is often an adequate substitute for factorization into primes. See http://cr.yp.to/coprimes.html for a bibliography.

For the rest of this section I'll focus on one important application, namely finding *small* factors of integers. The problem here is to find all the prime divisors of x below y, given x and y. This is a bottleneck in elliptic-curve primality proving (see http://cr.yp.to/bib/entries.html#2004/morain-ants), the number-field sieve (see the next section), et al.

It's easy to see this problem as a highly constrained example of factoring into coprimes, when y is small: if S contains all the primes up to y then factoring S into coprimes will reveal, for each element $x \in S$, the prime divisors of x below y. This view is traditionally ignored, because special-purpose techniques—Pollard's ρ method, for example, and Lenstra's elliptic-curve method—appear to be much faster. See http://cr.yp.to/bib/entries.html#1975/pollard and http://cr.yp.to/bib/entries.html#1987/lenstra-fiec.

I was surprised to realize a few years ago that my general algorithm to factor into coprimes had surpassed the special-purpose techniques. I showed in http://cr.yp.to/papers.html#sf that a streamlined algorithm takes time $(\lg y)^{3+o(1)}$ per input bit to find all prime divisors $\leq y$ of an integer having $(\lg y)^{O(1)}$ bits, if there are at least y integers to handle simultaneously. The best previous result was Lenstra's elliptic-curve method, which conjecturally takes time $\exp((\lg y)^{1/2+o(1)})$ per input bit.

Franke, Kleinjung, Morain, and Wirth recently introduced an algorithm—see Section 4 of http://cr.yp.to/bib/entries.html#preprint/franke—that takes time just $(\lg y)^{2+o(1)}$ per input bit to tell whether an integer having $(\lg y)^{O(1)}$ bits is y-smooth (i.e., has no prime divisors larger than y), if there are at least y integers to handle simultaneously. They described their algorithm as a "simplification" of mine; it *is* simpler, certainly, but it also has an extra idea that saves the $(\lg y)^{1+o(1)}$ factor. See http://cr.yp.to/papers.html#smoothparts for a slightly better algorithm.

The main bottleneck in these algorithms is computing $P \bmod x_1, P \bmod x_2, \ldots$, given $P, x_1, x_2, \ldots$. Building on previous work of Bostan, Lecerf, and Schost (see http://cr.yp.to/bib/entries.html#2003/bostan), I've recently introduced "scaled remainder trees," making this computation faster by a factor $2.6 + o(1)$. See http://cr.yp.to/papers.html#scaledmod.

At a lower level, the bottleneck in these algorithms is multiplication of large integers. The algorithms begin by computing the product of all the integers to be factored and the product of all the primes; these products are large even if all of the input integers are small. Every speedup in large-integer multiplication is a speedup in finding small factors of integers.

4 Factorization into primes

Sometimes one wants to factor an integer n into primes—not merely coprimes, and not merely small primes. The number-field sieve is conjectured to do this in time $L^{1.90\ldots+o(1)}$ where $L = \exp((\log n)^{1/3}(\log\log n)^{2/3})$. I've discovered several improvements in the number-field sieve:

- Searching for a polynomial: The number-field-sieve time depends on several parameters, notably a polynomial. A naive search for a good polynomial has (conjecturally) the same effect as subtracting $(2/5 + o(1)) \log T$ from $\log n$, where T is the total number-field-sieve time. The best search method in the literature, introduced by Murphy in 1999, replaces 2/5 with 2/3. I recently found a method that, using 4-dimensional lattice-basis reduction, replaces 2/3 with 6/7. See `http://cr.yp.to/talks.html#2004.11.15`, pages 10–18.
- Estimating polynomial size: I recently introduced a fast method to compute a good approximation to the distribution of small values of the polynomial by numerically approximating a superelliptic integral. The best previous method took time roughly the square root of the number of small values. See `http://cr.yp.to/talks.html#2004.11.15`, pages 4–9.
- Estimating smoothness probabilities: I introduced a fast method to compute tight bounds on the distribution of smooth elements of number fields. See `http://cr.yp.to/papers.html#psi`. This allows fast, accurate evaluation of the merit of a polynomial.
- Multiple lattices: In `http://cr.yp.to/papers.html#mlnfs` I analyzed the distribution of small polynomial values in the multiple-lattice number-field sieve, and in particular showed how to balance the lattices to find as many small values as possible.
- Recognizing smooth numbers: See the previous section. The textbook view is that this is more efficiently handled by sieving; the reality is that the fastest method combines sieving with the method of the previous section, as discussed in `http://cr.yp.to/talks.html#2004.06.14`.

I've started writing new high-speed factorization software. I'm going to publish it for everyone to use. (I haven't published much software in the last few years; my department colleagues have made clear to me that they don't value anything other than papers. Fortunately, my promotion has now been approved by the department and the dean, so I can return to a healthy balance between papers and software.)

Applications

Factorization into primes is a fundamental tool within computational number theory, but it is much more widely known as a tool to attack the most popular public-key cryptographic systems.

Users of public-key cryptographic systems of RSA/Rabin type are betting that factorization into primes is difficult. Each user has a "public key," a large

integer with secret factors; an attacker who can compute the secret factors can impersonate the user and snoop on the user's communications. Users want to choose small integers to save time, but users need to choose integers large enough to resist factorization. Is a 1024-bit integer safe? 1536? 2048? Suppose an attacker steals time on millions of computers around the Internet; how long will those computers take to factor a 1024-bit integer with state-of-the-art factorization software?

In my cryptographer's hat, I hope that there's a limit to how quickly we can factor integers into primes, and I hope that the number-field sieve is close to that limit. To the extent that the number-field sieve can be improved, it's important for us to find the improvements and warn users to move to larger keys that aren't vulnerable to attack.

Published high-speed factorization software is much more convincing than a mere estimate of factorization speed. Of course, the software will also be useful for number theorists who actually want to factor integers.

Price-performance ratio for circuits

A conventional computer of size $n^{1+o(1)}$ needs time $n^{1+o(1)}$ to sort n numbers, or to multiply two n-bit numbers. A mesh computer of size $n^{1+o(1)}$ can do the same jobs in time $n^{0.5+o(1)}$. There are many other calculations for which mesh architectures—in particular, circuits—are vastly more cost-effective than conventional computer architectures; this is the theme of thousands of articles in the very-large-scale-integration (VLSI) research literature.

I realized in September 2000 that the number-field sieve was one of these calculations. Specifically, a conventional number-field-sieve factorization (and Shamir's "TWINKLE"), as described in the literature, takes time $L^{1.901...+o(1)}$ on a machine of size $L^{0.950...+o(1)}$. (Subsequent price-performance-ratio optimization by Pomerance achieved time $L^{1.754...+o(1)}$ on a conventional computer of size $L^{1.006...+o(1)}$.) A mesh computer of size $L^{0.790...+o(1)}$ takes time just $L^{1.185...+o(1)}$. See `http://cr.yp.to/papers.html#nfscircuit`.

The next step is to pin down the $o(1)$. There are now several papers with various contradictory speculations regarding the speed of mesh factorization. The most amusing contrast is between a paper by Lenstra, Shamir, Tomlinson, and Tromer claiming (for absurd reasons) that mesh computers will have no effect on sieving speed for 1024-bit integers, and a paper by Shamir and Tromer claiming (by combining a series of wild guesses) that a ten-million-dollar mesh computer will handle sieving for 1024-bit integers in under a year. (See my web page `http://cr.yp.to/nfscircuit.html` for analysis of various Lenstra-Shamir-Tomlinson-Tromer errors, both technical and historical.)

In the meantime, I've been working on *verifiable* upper bounds: specifically, functioning factorization software for a mesh computer. I have convinced myself that 1024-bit integers can be factored at tolerable cost by a streamlined mesh implementation of the elliptic-curve method. It is important to realize that the crossover between the elliptic-curve method and the number-field sieve is much higher for mesh architectures than for conventional architectures.

5 Five more number-theoretic computations

Detecting perfect powers

`http://cr.yp.to/papers.html#powers` presents an algorithm that, given an integer $n > 1$, finds the largest integer k such that n is a kth power. The algorithm takes time $b^{1+o(1)}$ where b is the number of bits in n; more precisely, time $b \exp(O(\sqrt{\lg b \lg \lg b}))$; conjecturally, time $b(\lg b)^{O(1)}$. The run-time analysis relies on transcendental number theory. The best previous result, by Bach and Sorenson, was $b^{2+o(1)}$.

`http://cr.yp.to/papers.html#powers2` (with Lenstra and Pila) presents a newer algorithm for the same problem. This algorithm takes time $b(\lg b)^{O(1)}$. It relies on relatively complicated subroutines: specifically, on my fast algorithm to factor integers into coprimes.

Proving primality

`http://cr.yp.to/papers.html#quartic` presents an algorithm that, given a prime n, finds and verifies a proof of primality for n in random time $b^{4+o(1)}$. The algorithm improves on a line of work by Agrawal, Kayal, Saxena, Macaj, Berrizbeitia, and Cheng. See `http://cr.yp.to/papers.html#prime2004` for a picture of how this compares to previous results.

Enumerating small primes

`http://cr.yp.to/papers.html#primesieves` (joint with Atkin) presents an algorithm to enumerate the primes up to n using $O(n/\log \log n)$ additions and $n^{1/2+o(1)}$ bits of memory. The best previous results were $O(n)$ and $n^{1/2+o(1)}$, or $O(n/\log \log n)$ and $n^{1+o(1)}$, but not both simultaneously. Subsequent work by Galway improved 1/2 to 1/3.

Computing logarithms to high precision

`http://cr.yp.to/papers.html#logagm` presents an algorithm that computes b-bit logarithm bounds using $(5 + o(1)) \lg b$ multiplications, $(2 + o(1)) \lg b$ square roots, and a negligible amount of extra work. The best previous results were on the scale of $\lg b$ but with larger constants. All of these algorithms rely on the basic theory of elliptic integrals.

Enumerating locally square polynomial values

`http://cr.yp.to/papers.html#focus` presents an algorithm that identifies all the "pseudosquares" in a sequence of H small sieveable integers in average time $H^{1-(2+o(1))/\lg \log H}$ as $H \to \infty$. The usual method takes time $H^{1-(1+o(1))/\lg \log H}$; my algorithm is about 2^{10} times faster in practice. Williams and Wooding have used my algorithm to compute all pseudosquares up to 2^{80}; this computation allows fast proofs of primality for primes up to 2^{100}.

6 Secure private communication

Secret-key message authentication

I recently introduced Poly1305-AES, a state-of-the-art message-authentication code suitable for a wide variety of applications.

Given a variable-length message, a 16-byte nonce (unique message number), and a 32-byte secret key, Poly1305-AES computes a 16-byte authenticator. The security of Poly1305-AES is very close to the security of AES, the Advanced Encryption Standard: the security gap is below $14D\lceil L/16\rceil/2^{106}$ if messages have at most L bytes, the attacker sees at most 2^{64} authenticated messages, and the attacker attempts D forgeries. My Athlon implementation computes Poly1305-AES in fewer than $3.625(\ell + 170)$ cycles for an ℓ-byte message. See `http://cr.yp.to/papers.html#poly1305`.

There are two big advances here over the previous state of the art. First, previous high-speed functions use much larger inputs, typically several kilobytes per key, so they slow down dramatically in applications that handle many keys simultaneously. Second, previous security bounds for fast 16-byte authenticators broke down at a smaller number of messages, often below 2^{50}.

Designing Poly1305-AES had a somewhat different flavor from most of my work in computational number theory: I was searching not just for the best algorithm to compute a function, but for the best function to be computed. As usual, I found it helpful to build a mathematical framework for the literature. See Section 10 of `http://cr.yp.to/papers.html#hash127`.

Public-key secret sharing

In the Diffie-Hellman protocol, each user selects a secret key and a corresponding public key. Two users who see each other's public keys can each compute a shared secret suitable for use in (e.g.) the message-authentication code discussed above. Public keys have to be protected against forgery but don't have to be kept secret.

I've set speed records for the Diffie-Hellman protocol, specifically for elliptic-curve Diffie-Hellman, specifically for Diffie-Hellman on the standard "NIST P-224" elliptic curve selected by Jerry Solinas at NSA. My software is under 1000000 clock cycles on most popular computers; it's between 1.6 and 6 times faster than the previous records for NIST P-224, and far faster than comparable uses of RSA.

Technology transfer: I've put a lot of effort into making this software easy to use. My web pages `http://cr.yp.to/nistp224.html` contain summaries of why the software is interesting; the software itself, with streamlined installation instructions; details of the command-line interface; details of the C interface; an overview of the machine-specific optimizations in the software; detailed timings for many machines; a discussion of the raw floating-point performance of each machine; an analysis of how much the software speed could be improved; a list of twenty-two previous results; a discussion of the security of cryptographic applications; and an analysis of seven specific patents (none applicable).

I can achieve somewhat better speeds using the carefully selected curve $y^2 = x^3 + 7530x^2 + x$ modulo $2^{226} - 5$. I'm continuing to look for speedups in elliptic-curve and hyperelliptic-curve cryptography. I'm also planning to rewrite my software using `floatasm`.

Protecting the Internet

Cryptography on the Internet is in a sorry state. Every serious Internet protocol now has some sort of "security" extension, but only a tiny fraction of data on the Internet is actually cryptographically protected. Sometimes the security extensions are too hard to use. Sometimes the extensions are usable but too slow for busy servers.

I'm adding link-level cryptographic protection to the Internet mail system. For each message, the SMTP client looks up the SMTP server's public key in DNS (which will be protected as discussed in the next section), computes a shared secret using elliptic-curve Diffie-Hellman, and uses the shared secret to authenticate every packet using Poly1305-AES. Unlike existing mail "security" protocols, this will be fast enough to use for every message, and it will protect against attackers actively misdirecting messages.

7 Secure public communication

Public-key signatures

`http://cr.yp.to/sigs.html` defines a state-of-the-art Rabin-type public-key signature system. Each user of the system has a 64-byte public key and a 512-byte secret key. A user can quickly generate a 193-byte signature of a message. Anyone can very quickly verify that the signature matches the message and the user's public key. Signatures can be expanded to 385 bytes for even faster verification (typically under 10000 clock cycles), or compressed to 97 bytes at some cost in verification time; anyone can perform this compression and expansion on the fly, adapting signatures to the local costs of bandwidth and CPU time.

It appears to be extremely difficult for anyone other than the user to sign messages under the user's public key. `http://cr.yp.to/papers.html#rwtight` proves that any *generic* attack on the signature system—any attack that works for all hash functions, or for a noticeable fraction of all hash functions—can be converted into a comparably fast algorithm to factor 1536-bit integers. Of course, there might be a fast factorization algorithm, and there might be a fast non-generic attack on the particular hash function in use, but this is the best type of security guarantee we have for any public-key signature system offering fast verification.

My software isn't ready for release yet. One reason is that I'm building a new hash function from scratch. I was using SHA-1, but I'd like to use something bigger after Wang's discovery of MD5 collisions. I feel safe with SHA-256, but SHA-256 is so slow that it has a quite noticeable impact on signature-verification time.

Protecting the Internet

The Domain Name System is a distributed database that maps domain names, such as `cr.yp.to`, to IP addresses (and other data), such as `131.193.178.160`.

Today's DNS records are not cryptographically protected. By giving fake IP addresses to your computer, an attacker can easily steal your incoming and outgoing mail, substitute fake web pages, etc.

DNSSEC, a project to cryptographically sign DNS records, has been under development for ten years by a large (and obviously rather incompetent) team of protocol developers. I've designed a new DNS security system, DNSSEC2, with several advantages over DNSSEC: it's faster; it uses much stronger cryptographic tools; it protects against denial of service; it allows end-to-end security without help from the centralized root servers; and, most importantly, it works with existing DNS database software, so it's much easier than DNSSEC to deploy. See `http://cr.yp.to/talks.html#2004.04.28`. I plan to release the DNSSEC2 software this year.

8 Anti-bug programming environments

All the cryptographic protection in the world won't stop an attacker who exploits a bug, such as a buffer overflow, to seize control of your computer. Some of the most widely deployed programs on the Internet—the BIND DNS server, for example, and the Sendmail SMTP server (post-office program)—have had *thousands* of bugs, including many bugs that allowed attackers to seize control.

For more than a decade I have been systematically identifying error-prone programming habits—by reviewing the literature, by analyzing other people's mistakes, and by analyzing my own mistakes—and redesigning my programming environment to eliminate those habits. For example, "escape" mechanisms, such as backslashes in various network protocols and `%` in `printf`, are error-prone: it's too easy to feed "normal" strings to those functions and forget about the escape mechanism.[1] I switched long ago to explicit tagging of "normal" strings; the resulting APIs are wordy but no longer error-prone. The combined result of many such changes has been a drastic reduction in my own error rate.

Starting in 1997, I offered $500 to the first person to publish a verifiable security hole in the latest version of `qmail`, my Internet-mail transfer agent; see `http://cr.yp.to/qmail/guarantee.html`. There are now more than a million Internet SMTP servers running `qmail`. Nobody has attempted to claim the $500.

Starting in 2000, I made a similar $500 offer for `djbdns`, my DNS software; see `http://cr.yp.to/djbdns/guarantee.html`. This program is now used to publish the IP addresses for two million `.com` domains: `citysearch.com`, for example, and `lycos.com`. Nobody has attempted to claim the $500.

[1] Format-string bugs were not widely appreciated until 2000; see the introduction to `www.usenix.org/publications/library/proceedings/sec01/shankar.html`. Four years before that, I had realized and publicly pointed out the security impact of a format-string bug in `logger`, a standard UNIX-command-line interface to `syslog`; see `www.ornl.gov/lists/mailing-lists/qmail/1996/12/msg00314.html`.

There were several non-security bugs in `qmail`, and a few in `djbdns`. My error rate has continued to drop since then. I'm no longer surprised to whip up a several-thousand-line program and have it pass all its tests the first time.

Bug-elimination research, like other user-interface research, is highly non-mathematical. The goal is to have users, in this case programmers, make as few mistakes as possible in achieving their desired effects. We don't have any way to model this—to model human psychology—except by experiment. We can't even *recognize* mistakes without a human's help. (If you can write a program to recognize a class of mistakes, great—we'll incorporate your program into the user interface, eliminating those mistakes—but we still won't be able to recognize the *remaining* mistakes.) I've seen many mathematicians bothered by this lack of formalization; they ask nonsensical questions like "How can you prove that you don't have any bugs?" So I sneak out of the department, take off my mathematician's hat, and continue making progress towards the goal.

9 Extreme sandboxing

The software installed on my newest home computer contains 78 million lines of C and C++ source code. Presumably there are hundreds of thousands of bugs in this code. Removing all of those bugs would be quite expensive. Fortunately, there's a less expensive way to eliminate most security problems.

Consider, for example, a compressed music track. The UNIX `ogg123` program and underlying libraries contain 50000 lines of code whose sole purpose is to read a compressed music track in Ogg Vorbis format and write an uncompressed music track in wave format.

UNIX makes it possible (though unnecessarily difficult) to build a `safeogg` program that does the same conversion *and that has no other power over the system*. Bugs in the 50000 lines of code are then irrelevant to security: if the input is from an attacker who seizes control of `safeogg`, the most the attacker can do is write arbitrary output, which is what the input source was authorized to do anyway.

How does `safeogg` eliminate all other access to the system? It eliminates its access to the filesystem by using `chroot`. It eliminates its access to outside RAM by acquiring a temporary uid not shared with any other processes; this takes some effort but is doable. It eliminates its access to the network by setting a hard `nofile` resource limit of 0. It limits its own use of RAM by setting other resource limits. A completely different way to accomplish the same result—to build an "extreme sandbox" for `safeogg`—is to run those 50000 lines of code inside an interpreter that imposes the same constraints.

The existing `ogg123` program can be converted into a small wrapper around `safeogg`. Similarly, any other program that wants to uncompress music tracks can be restructured to delegate the job to `safeogg`. This requires a small amount of programming effort—much less than eliminating all the bugs in those 50000 lines of code.

The relevant property of the `safeogg` transformation is that the output isn't trusted any more highly than any of the inputs. This is a simple statement of the security policy for music tracks: everyone authorized to affect the input, a compressed music track, is also authorized to have complete control over the output, an uncompressed music track.

Any transformation with the same property becomes irrelevant to security if the system is restructured to place that transformation into an extreme sandbox.

Do *all* transformations have this property? Of course not. There are some transformations that produce output trusted more highly than the input; these transformations have to be bug-free. For example, when mail messages from Alice and Bob are delivered to a user's mailbox, the output (the mailbox file or directory) is trusted to contain a correct copy of Alice's message, even though the output also depends on input from Bob, who isn't trusted with Alice's message; and vice versa. The code that separates messages in mailboxes is security-critical; it has to be bug-free. (There are also a few transformations that are run so often that putting them into an extreme sandbox would impose an unacceptable speed penalty.)

But these exceptions account for only a small fraction of the code in a system. For example, compared to the code that separates messages in mailboxes, there is vastly more code that transforms individual messages; the latter code can run in an extreme sandbox and doesn't have to be bug-free.

I should emphasize here that there's a fundamental difference between this project and typical sandboxing projects described in the literature. The goal of a typical sandboxing project is to apply as many restrictions as possible to a program while receiving no help from the programmer; the problem is that this doesn't stop all attacks.[2] In contrast, I insist on an extreme sandbox guaranteeing complete security, and I then ask how the programmer's time can be minimized. As in Section 1, programs are not static objects; the programmer *cooperates* with the sandboxing tools.

I expect this strategy to produce invulnerable computer systems: restructure programs to put almost all code into extreme sandboxes; eliminate bugs in the small volume of remaining code. I won't be satisfied until I've put the entire security industry out of work.

[2] `www.usenix.org/publications/library/proceedings/sec96/goldberg.html`, for example, described a sandboxing tool that applied *some* restrictions to Netscape's "DNS helper" program. The subsequently discovered `libresolv` bug was a security hole in that program despite the sandbox. Imposing heavier restrictions would have meant changing the program.