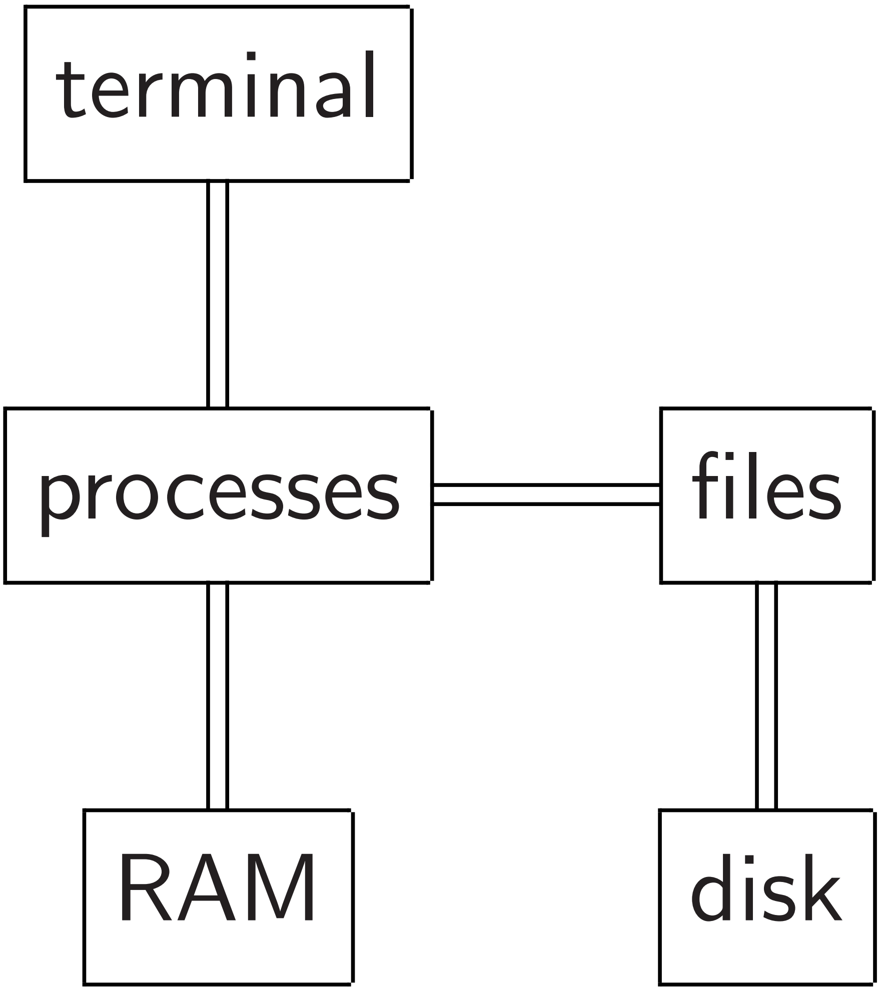
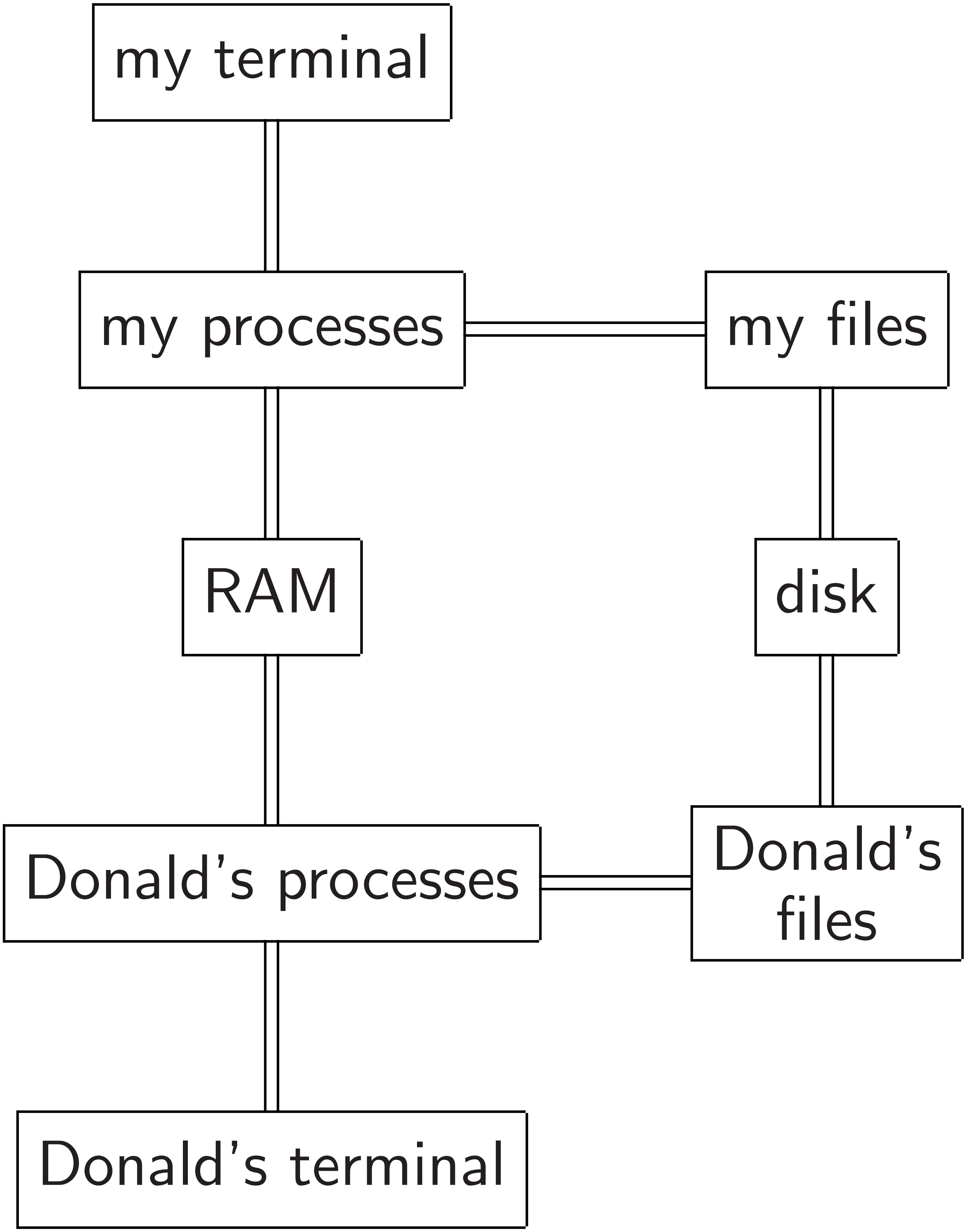
Usable verification of fast cryptographic software

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven



Operating-system kernel
divides RAM among processes,
divides disk among files.
Provides convenient functions
for processes to access files,
start new processes, etc.


my terminal
my processes
my files
RAM
disk
Donald's processes
Donald's files
Donald's terminal

Can Donald corrupt the data appearing on my terminal?

Can Donald corrupt the data appearing on my terminal?

Attack: guess my password.

Can Donald corrupt the data appearing on my terminal?

Attack: guess my password.
Defense: I have a high-entropy randomly generated password.

Can Donald corrupt the data appearing on my terminal?

Attack: guess my password.
Defense: I have a high-entropy randomly generated password.

Attack: replace the terminal with a rigged terminal that intercepts my password.

Can Donald corrupt the data appearing on my terminal?

Attack: guess my password.
Defense: I have a high-entropy randomly generated password.

Attack: replace the terminal with a rigged terminal that intercepts my password.
Defense: physical security.

Can Donald corrupt the data appearing on my terminal?

Attack: guess my password.
Defense: I have a high-entropy randomly generated password.

Attack: replace the terminal with a rigged terminal that intercepts my password.
Defense: physical security.

Attack: use my terminal earlier and leave a program running that looks like the usual login screen but intercepts my password.

Can Donald corrupt the data appearing on my terminal?

Attack: guess my password.
Defense: I have a high-entropy randomly generated password.

Attack: replace the terminal with a rigged terminal that intercepts my password.
Defense: physical security.

Attack: use my terminal earlier and leave a program running that looks like the usual login screen but intercepts my password.
Defense: secure attention key.

Donald is authorized to store data on the same computer.

Attack: Donald stores data in my part of RAM, or my part of disk.

Donald is authorized to store data on the same computer.

Attack: Donald stores data in my part of RAM, or my part of disk.

Two-part defense:

1. "Memory protection". Hardware does not allow processes to access data outside areas marked by kernel.

2. Kernel keeps track of which parts of RAM and disk are mine, and which parts are Donald's.

Bugs in this kernel code
can compromise security,
allowing Donald to write
to my part of RAM or disk.

Bugs in this kernel code
can compromise security,
allowing Donald to write
to my part of RAM or disk.

Fix: Eliminate the bugs!

Bug-free code is expensive
but not impossible when
code volume is small enough.

Successful example:
computer-verified proof of
seL4 microkernel correctness,
including RAM partitioning etc.

If a small bug-free kernel has cut off Donald's communication with me:

I can run a 10000000-line program filled with bugs, and still be confident that Donald is unable to corrupt the output of the program.

If a small bug-free kernel
has cut off Donald's
communication with me:

I can run a 10000000-line
program filled with bugs,
and still be confident that
Donald is unable to corrupt
the output of the program.

The **trusted computing base**
(TCB) is the part of the system
that enforces security policy.
The 10000000-line program
is not part of the TCB.

But we *want* communication!

Today: Alice sends me email.
I download Bob's web page.
These users are authorized
to put data on my screen.

But we *want* communication!

Today: Alice sends me email.
I download Bob's web page.
These users are authorized
to put data on my screen.

Security policy: Whenever the
computer shows me a file, it also
tells me the source of the file.

But we *want* communication!

Today: Alice sends me email.
I download Bob's web page.
These users are authorized
to put data on my screen.

Security policy: Whenever the
computer shows me a file, it also
tells me the source of the file.

If Donald creates a file
and convinces the computer
to show me the file
as having source "Alice"
then this policy is violated.

PRN Pwn2Own 2016: Chin... ×

www.prnewswire.com/news-releases/pwn2own-2

PR Newswire

Pwn2Own 2016: Chinese Researcher Hacks Google Chrome within 11 minutes

Mar 17, 2016, 09:12 ET from Qihoo 360

Chinese Security Team in Global Arena

Facebook Twitter Pinterest

VANCOUVER, British Columbia, March 17, 2016 /PRNewswire/ -- 360Vulcan Team from Qihoo 360 hacked Google Chrome, the browser with the least vulnerabilities, and obtained the highest system privilege. It's the first time a Chinese security team has hacked Google Chrome at the Pwn2Own contest.

360Vulcan Team also hacked Adobe Flash Player based on Edge browser, obtaining the highest system privilege, which won the team a USD 80,000

Which part of the system enforces the security policy?

Which part of the system
enforces the security policy?

Widely deployed software systems
make no real efforts to limit this.

There is some "security" code
inside kernel and browser.
But bugs in other code
can and do compromise security.
TCB has $>$30000000 lines.

Which part of the system
enforces the security policy?

Widely deployed software systems
make no real efforts to limit this.

There is some “security” code
inside kernel and browser.
But bugs in other code
can and do compromise security.
TCB has >30000000 lines.

Fix: rearchitect entire system
so that a **small** TCB
tracks sources of all data.
Eliminate all bugs in TCB.

Cryptography in the TCB

What happens if data is sent through Donald's network?

Cryptography in the TCB

What happens if data is sent through Donald's network?

Solution: Sender and receiver scramble communication in a way that Donald cannot understand and cannot silently corrupt.

OpenSSL crypto library has 500000 lines of code, and there are many other crypto libraries.

All of this is in the TCB.
Many devastating security bugs.

Why is crypto so big?

OpenSSL crypto library has 500000 lines of code, and there are many other crypto libraries.

All of this is in the TCB.
Many devastating security bugs.

Why is crypto so big?

Most important answer:
the pursuit of performance.

(Same issue elsewhere in TCB, but most blatant for crypto.
The rest of this talk
will focus on crypto.)

e.g. Variable-length-big-integer arithmetic library inside OpenSSL consumes 50000 lines of code. Includes 38 asm implementations optimized for various CPUs.

e.g. Variable-length-big-integer arithmetic library inside OpenSSL consumes 50000 lines of code. Includes 38 asm implementations optimized for various CPUs.

e.g. ECDSA signature verification: $(H(M)/S)B + (x(R)/S)A = R$, with S checked to be nonzero.

OpenSSL has complicated code for fast computation of $1/S$.

Checking $H(M)B + x(R)A = SR$ would be somewhat slower.

e.g. NIST P-256 prime p is
$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

ECDSA standard specifies reduction procedure given an integer "A less than p^2":

Write A as
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, A_{10}, A_9,$
$A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$,
meaning $\sum_i A_i 2^{32i}$.

Define
$T; S_1; S_2; S_3; S_4; D_1; D_2; D_3; D_4$
as

$(A_7, A_6, A_5, A_4, A_3, A_2, A_1, A_0)$;
$(A_{15}, A_{14}, A_{13}, A_{12}, A_{11}, 0, 0, 0)$;
$(0, A_{15}, A_{14}, A_{13}, A_{12}, 0, 0, 0)$;
$(A_{15}, A_{14}, 0, 0, 0, A_{10}, A_9, A_8)$;
$(A_8, A_{13}, A_{15}, A_{14}, A_{13}, A_{11}, A_{10}, A_9)$;
$(A_{10}, A_8, 0, 0, 0, A_{13}, A_{12}, A_{11})$;
$(A_{11}, A_9, 0, 0, A_{15}, A_{14}, A_{13}, A_{12})$;
$(A_{12}, 0, A_{10}, A_9, A_8, A_{15}, A_{14}, A_{13})$;
$(A_{13}, 0, A_{11}, A_{10}, A_9, 0, A_{15}, A_{14})$.

Compute $T + 2S_1 + 2S_2 + S_3 + S_4 - D_1 - D_2 - D_3 - D_4$.

Reduce modulo p "by adding or subtracting a few copies" of p.

Next-generation crypto

One of my favorite topics: removing tensions between security, simplicity, speed.

In particular, designing simple high-security crypto setting new speed records.

e.g. 2006 Bernstein "Curve25519" is twice as fast as standard ECC and much simpler to implement.

$>$1000000000 Curve25519 users today: iOS, Signal, OpenSSH, Tor, QUIC, WhatsApp, more.

NaCl: fast easy-to-use high-security crypto library. Joint work with Lange and Schwabe.

`nacl.cr.yp.to`

NaCl: fast easy-to-use high-security crypto library. Joint work with Lange and Schwabe.
nacl.cr.yp.to

TweetNaCl: self-contained 100-tweet C library providing the same easy-to-use high-security functions. Joint work with van Gastel, Janssen, Lange, Schwabe, Smetsers.
twitter.com/tweetnacl

NaCl: fast easy-to-use high-security crypto library. Joint work with Lange and Schwabe.
nacl.cr.yp.to

TweetNaCl: self-contained 100-tweet C library providing the same easy-to-use high-security functions. Joint work with van Gastel, Janssen, Lange, Schwabe, Smetsers.
twitter.com/tweetnacl

Can we guarantee zero bugs in TweetNaCl? And in NaCl?

Biggest challenge: the gap between big-integer operations such as $a, b \mapsto ab \bmod 2^{255} - 19$ and (e.g.) 32-bit operations.

Biggest challenge: the gap between big-integer operations such as $a, b \mapsto ab \bmod 2^{255} - 19$ and (e.g.) 32-bit operations.

Some big-integer software has been formally verified. Could NaCl switch to this?

Biggest challenge: the gap between big-integer operations such as $a, b \mapsto ab \bmod 2^{255} - 19$ and (e.g.) 32-bit operations.

Some big-integer software has been formally verified. Could NaCl switch to this?

1. Not state-of-the-art speed. Okay for TweetNaCl; not NaCl.

Biggest challenge: the gap between big-integer operations such as $a, b \mapsto ab \bmod 2^{255} - 19$ and (e.g.) 32-bit operations.

Some big-integer software has been formally verified. Could NaCl switch to this?

1. Not state-of-the-art speed. Okay for TweetNaCl; not NaCl.

2. Input-dependent timing. Timing can leak secret keys. Not okay even for TweetNaCl.

ACM CCS 2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang "Verifying Curve25519 software": computer-aided proof of correctness of main loops in two high-speed asm Curve25519 implementations.

ACM CCS 2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang "Verifying Curve25519 software": computer-aided proof of correctness of main loops in two high-speed asm Curve25519 implementations.

Proof required extensive human effort for each implementation: many detailed annotations, plus higher-level composition work.

ACM CCS 2014 Chen–Hsu–Lin–Schwabe–Tsai–Wang–Yang–Yang "Verifying Curve25519 software": computer-aided proof of correctness of main loops in two high-speed asm Curve25519 implementations.

Proof required extensive human effort for each implementation: many detailed annotations, plus higher-level composition work.

Each proof also required many hours of computer time.

Joint work with Schwabe:
new verifier `gfverif`
focusing on arithmetic mod p.
`gfverif.cryptojedi.org`

Automatically build computation graph from original code.

Automatically analyze ranges, convert ops into polynomials.
New peephole range optimizer.

Ask human for occasional annotations expressing high-level computations on integers mod p.

Have verified entire Curve25519 computation, not just main loop, for another implementation.

Only 1 minute of computer time.

Under 300 lines of easy annotations per implementation.

Usable by crypto developers.

Continuing to improve `gfverif` annotation language. Should be able to reduce below 100 annotations per implementation.