



# Assuring Crypto Code with Automated Reasoning

Aaron Tomb

Galois, Inc.

QCon, London

March 8, 2017



OpenSSL Security Advisory [07 Apr 2014]

TLS heartbeat read overrun (CVE-2014-0160)

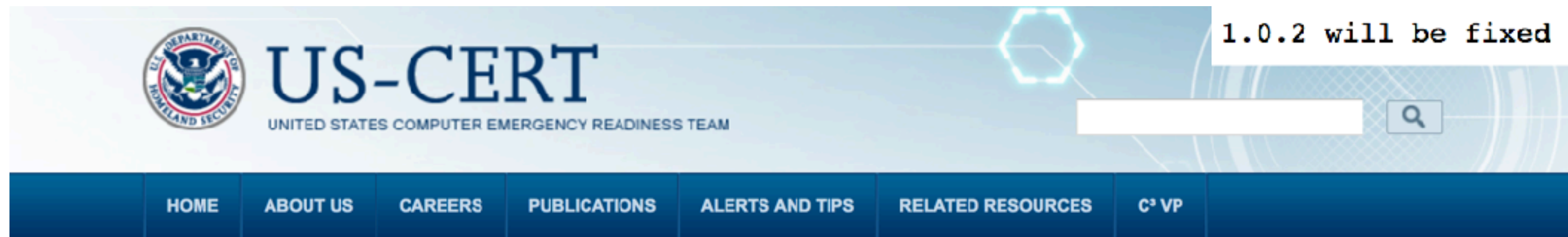
A missing bounds check in the handling of the TLS heartbeat extension can be used to reveal up to 64k of memory to a connected client or server.

Only 1.0.1 and 1.0.2-beta releases of OpenSSL are affected including 1.0.1f and 1.0.2-beta1.

Thanks for Neel Mehta of Google Security for discovering this bug and to Adam Langley <agl@chromium.org> and Bodo Moeller <bmoeller@acm.org> for preparing the fix.

Affected users should upgrade to OpenSSL 1.0.1g. Users unable to immediately upgrade can alternatively recompile OpenSSL with `-DOPENSSL_NO_HEARTBEATS`.

1.0.2 will be fixed in 1.0.2-beta2.



**Alert (TA14-098A)**

OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160)

Original release date: April 08, 2014 | Last revised: October 05, 2016

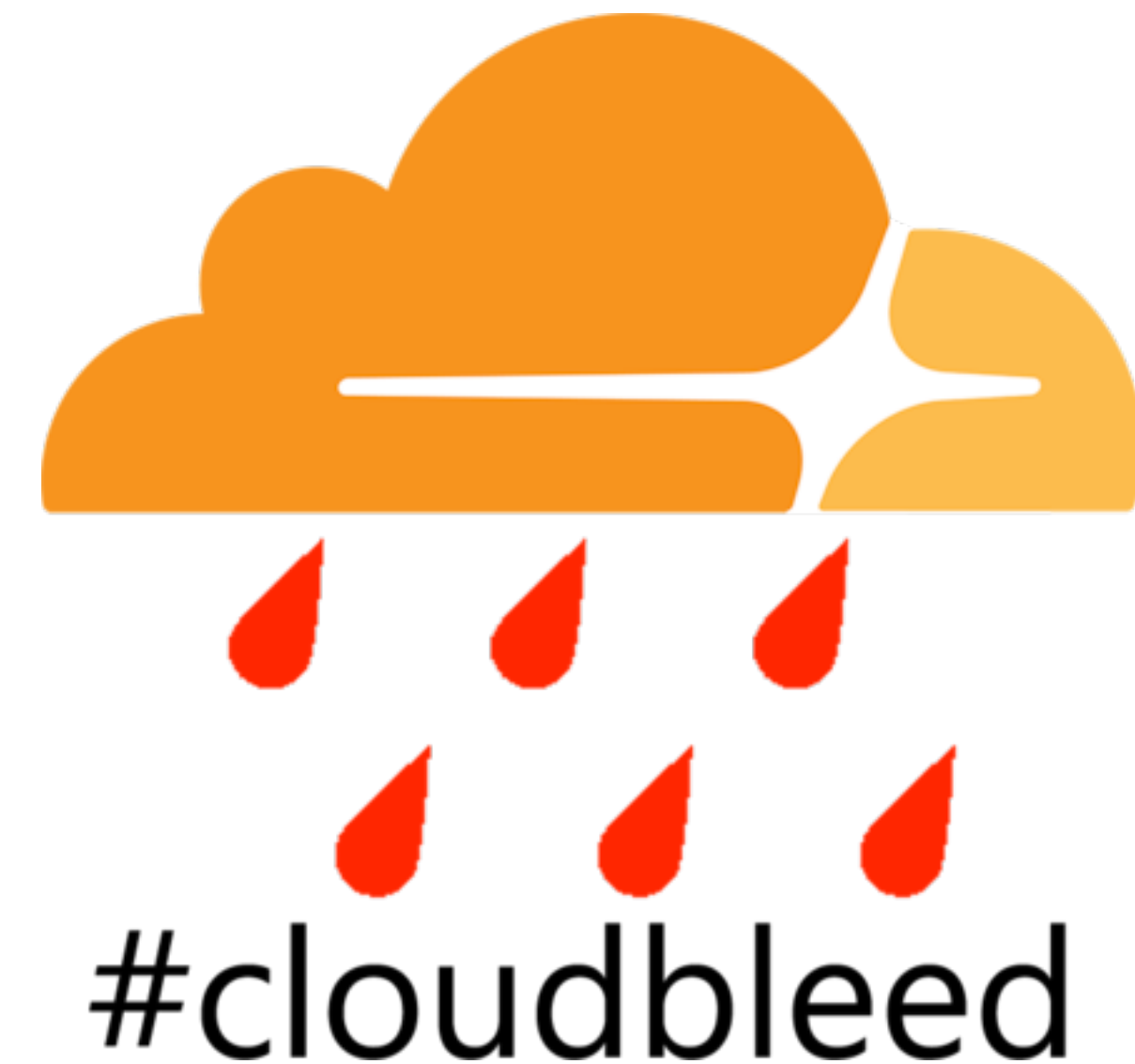
Print Tweet Send Share

**Systems Affected**

- OpenSSL 1.0.1 through 1.0.1f
- OpenSSL 1.0.2-beta

**Overview**

A vulnerability in OpenSSL could allow a remote attacker to expose sensitive data, possibly including user authentication credentials and secret keys, through incorrect memory handling in the TLS heartbeat extension.



# Generic Flaws

- Huge impact, but...
- Generic misuse of language

```
if ( ++p == pe )  
    goto _test_eof;
```

- No need to understand intention of code
- Visibility helps with quick remediation



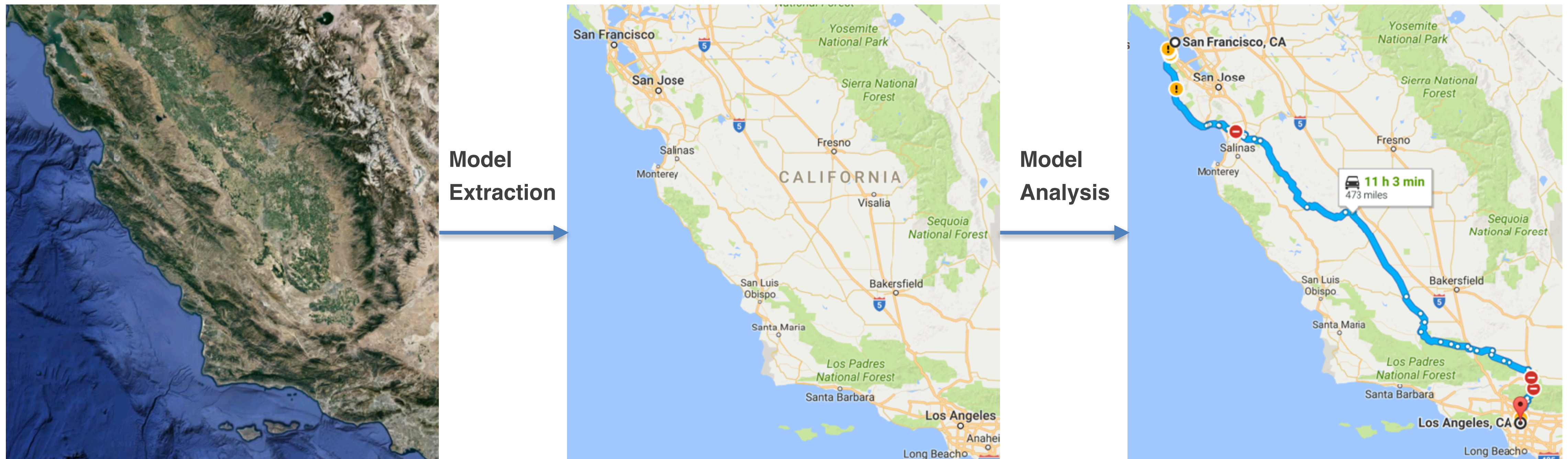
# Why Is This Bug Interesting?

- Couldn't be detected by generic tool
  - Need to know what the code should do!
- Discovered day before 0.9.8g release; fixed 6mo later
  - Many users didn't upgrade quickly
  - Exploit described 4 years later

# Introduction

- R&D Lead @ Galois
  - Focused on software correctness
- Developing tools to check that code does what's *intended*
  - With high confidence about all possible inputs
- This talk: the Software Analysis Workbench (SAW)
  - Open source tool with a high degree of automation

# Can I get from SF to LA?



**Moving from the territory to the map is possible for software!**

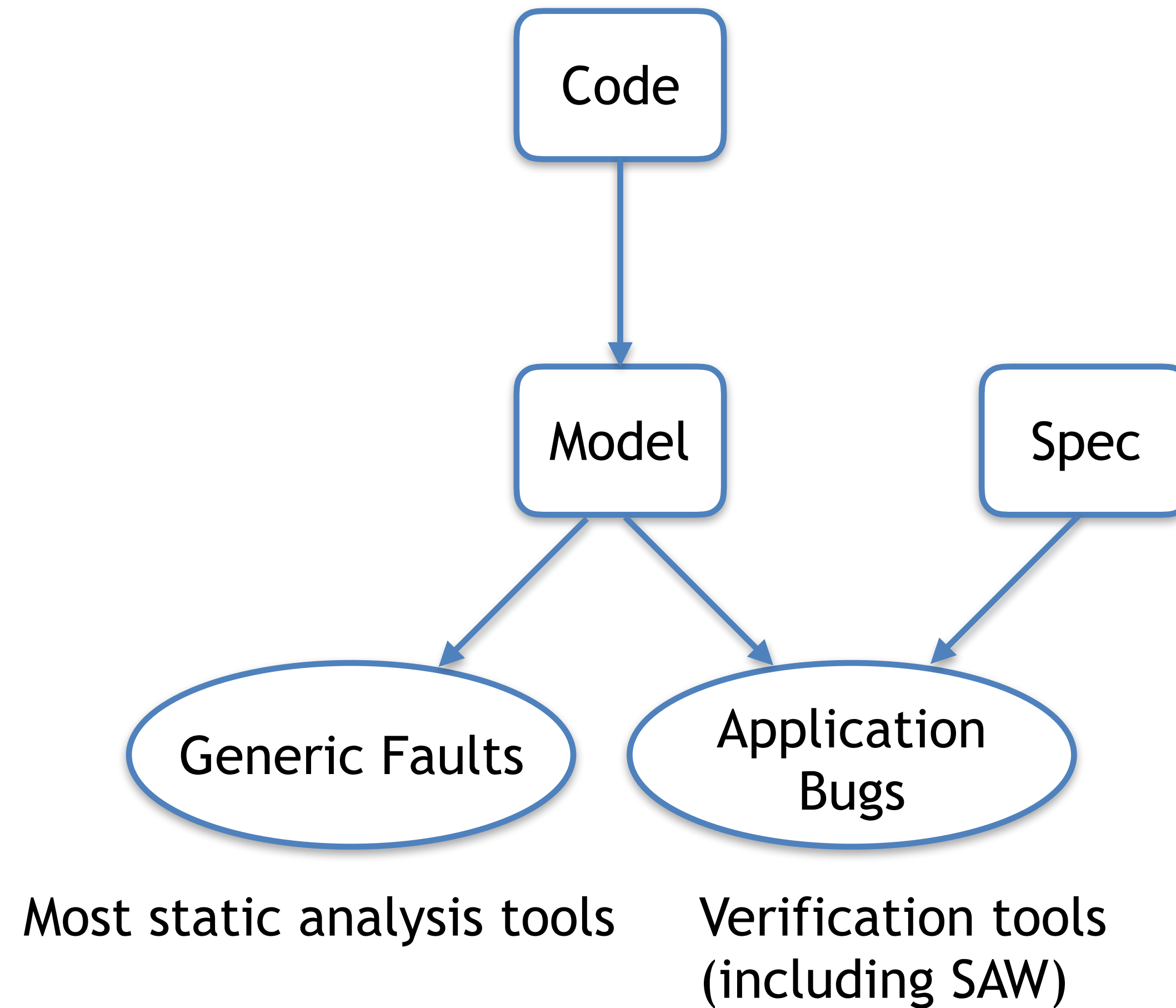
Imagery ©2017 Data SIO, NOAA, U.S. Navy, NGA, GEBCO, Landsat / Copernicus, Data LDEO-Columbia, NSF, NOAA, Map data ©2017 Google, INEGI

# Specifications and Implementations

- Implementation: CA itself
- Specification: driving from SF to LA is (always) possible
- Map of CA is model of CA, or more detailed specification
  - Model: things that **can** be done
  - Spec: things that **should** be done
  - Represented in the same way! Therefore, **comparable**.



# Generic vs. Application-Specific Bugs



## Reference and Optimized “Find First Set”

```
uint32_t ffs1(uint32_t w) {
    int cnt, i = 0;
    if(!w) return 0;
    for(cnt = 0; cnt < 32; cnt++)
        if((1 << i++) & w)
            return i;
    return 0;
}
```

```
uint32_t ffs2(uint32_t w) {
    uint32_t r, n = 1;
    if(!(w & 0xffff))
        { n += 16; w >>= 16; }
    if(!(w & 0x00ff))
        { n += 8; w >>= 8; }
    if(!(w & 0x000f))
        { n += 4; w >>= 4; }
    if(!(w & 0x0003))
        { n += 2; w >>= 2; }
    r = (n + ((w + 1) & 0x01));
    return (w) ? r : 0;
}
```

# Testing to Compare Specs and Implementations

```
int ffs_test(uint32_t w) {  
    return ffs1(w) == ffs2(w);  
}
```

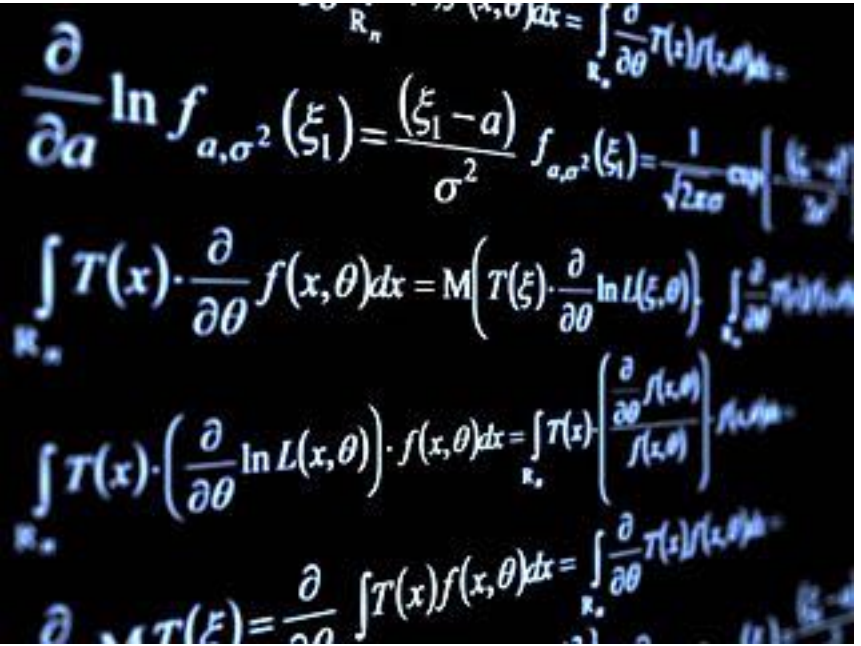
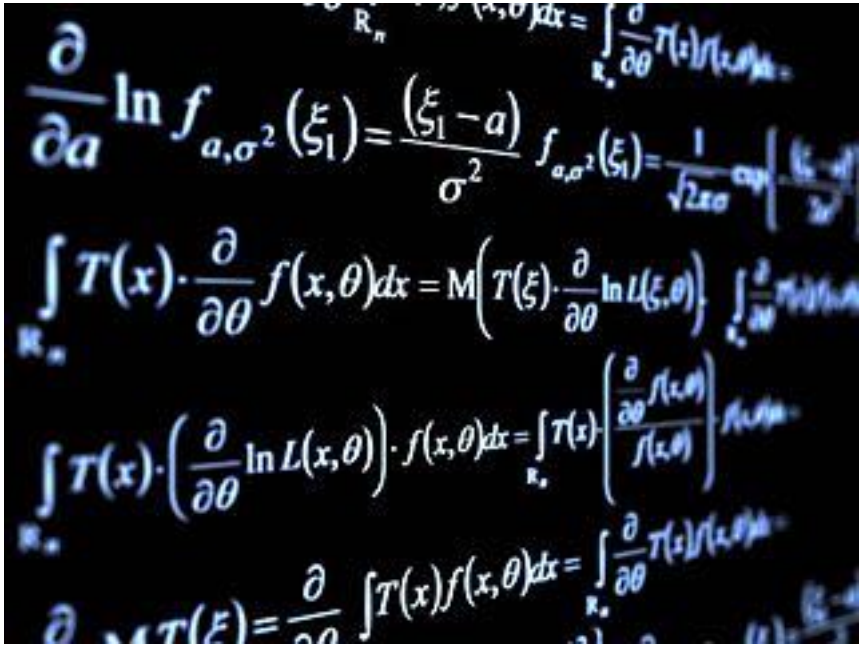
- Could run on carefully-chosen values
- Could run on many randomly-chosen values
- In the map metaphor: “I once drove from SF to LA, and it went fine.”

# Exhaustive Testing Would be Ideal!

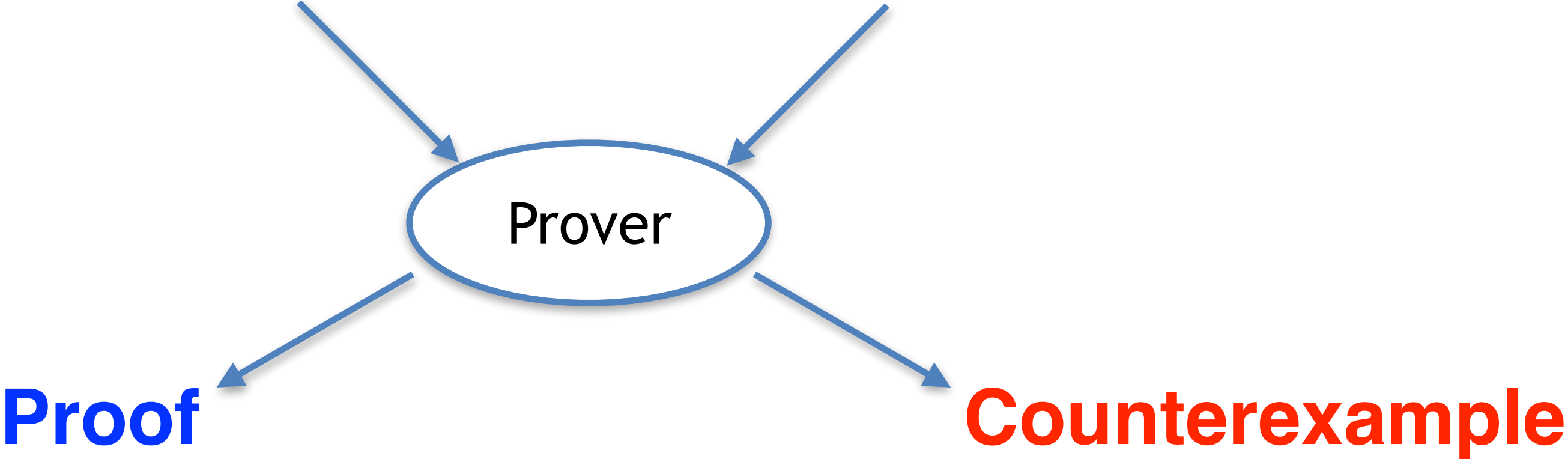
- Could maybe **exhaustively** test `ffs`, but only just
  - And it's trivial code (64-bit version intractable)
- Most code is much bigger, and wouldn't be tractable
- In the map metaphor: "I tried to get from SF to LA using **every possible** vehicle and succeeded."
  - How long would this take to do?

# Exhaustive Testing via Automated Reasoning

```
uint32_t ffs1(uint32_t w) {
    int cnt, i = 0;
    if(!w) return 0;
    for(cnt = 0; cnt < 32; cnt++)
        if((1 << i++) & w)
            return i;
    return 0;
}
```



```
uint32_t ffs2(uint32_t w) {
    uint32_t r, n = 1;
    if(!(w & 0xffff))
        { n += 16; w >>= 16; }
    if(!(w & 0x00ff))
        { n += 8; w >>= 8; }
    if(!(w & 0x000f))
        { n += 4; w >>= 4; }
    if(!(w & 0x0003))
        { n += 2; w >>= 2; }
    r = (n + ((w + 1) & 0x01));
    return (w ? r : 0);
}
```



- Map metaphor: “By applying graph analysis to the map, I know it’s always possible to get from NY to LA, whatever weight, height, or vehicle type”

# The Software Analysis Workbench (SAW)

- Extracts **models** from programs
  - Supports common languages through JVM, LLVM
  - Most used for Java and C, works with some Rust, C++, others
- Transforms, compares, and **proves things** about models
- Builds on powerful **automated reasoning** technology
  - SAT
  - SMT
  - Manual rewriting

# Proving FFS Equivalence

```
ffs_llvm.saw
```

```
l      <- llvm_load_module "ffs.bc";
ffs_ref <- llvm_extract l "ffs1" llvm_pure;
ffs_imp <- llvm_extract l "ffs2" llvm_pure;
let thm1 = {{ ffs_ref === ffs_imp }};
result  <- time (prove abc thm1);
print result;
```

```
$ saw ffs_llvm.saw
Loading module Crypto1
Loading file "ffs_llvm.saw"
Time: 0.024025s
Valid
```

## 64-bit “Find First Set”

```
uint64_t ffs1(uint64_t w) {  
    int cnt, i = 0;  
    if(!w) return 0;  
    for(cnt = 0; cnt < 64; cnt++)  
        if((1 << i++) & w)  
            return i;  
    return 0;  
}
```

```
$ saw ffs64_llvm.saw  
Loading module Crypto1  
Loading file "ffs64_llvm.saw"  
Time: 0.02996s  
Invalid: [w = 0x8000000000000000]
```



## 64-bit “Find First Set”

```
uint64_t ffs1(uint64_t w) {  
    int cnt, i = 0;  
    if(!w) return 0;  
    for(cnt = 0; cnt < 64; cnt++)  
        if(((uint64_t)1) << i++ & w)  
            return i;  
    return 0;  
}
```

```
$ saw ffs64_llvm_fixed.saw  
Loading module Cryptol  
Loading file "ffs64_llvm_fixed.saw"  
Time: 0.053556s  
Valid
```

# Dealing with Pointers

```
void  
swap_xor(uint8_t *x, uint8_t *y) {  
    *x = *x ^ *y;  
    *y = *x ^ *y;  
    *x = *x ^ *y;  
}
```

```
let swap_spec = do {  
    x <- fresh_var "x" (llvm_int 8);  
    y <- fresh_var "y" (llvm_int 8);  
    xp <- alloc (llvm_int 8);  
    yp <- alloc (llvm_int 8);  
    points_to xp (term x);  
    points_to yp (term y);  
  
    execute_func [xp, yp];  
  
    points_to xp (term y);  
    points_to yp (term x);  
};
```

```
load_llvm_module "swap_xor.bc";  
llvm_verify "swap_xor" [] swap_spec;
```

# What Makes Crypto Code Tractable?

- **Short** code, typically
- Small, fixed **input** and **output sizes**
- Constrained execution time
- **Specifications** exist

Ultimately, decidable (avoids halting problem)

# Cryptographic Specifications

**FIPS PUB 180-4**

---

**FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION**

**Secure Hash Standard (SHS)**

---

CATEGORY: COMPUTER SECURITY    SUBCATEGORY: CRYPTOGRAPHY

---

Information Technology Laboratory  
National Institute of Standards and Technology  
Gaithersburg, MD 20899-8900

This publication is available free of charge from:  
<http://dx.doi.org/10.6028/NIST.FIPS.180-4>

August 2015



U.S. Department of Commerce  
*Penny Prutzan, Secretary*

National Institute of Standards and Technology  
*Willie E. May, Under Secretary for Standards and Technology and Director*

Federal Information  
Processing Standards Publication 197

November 26, 2001

**Announcing the**

**ADVANCED ENCRYPTION STANDARD (AES)**

Federal Information Processing Standards Publications (FIPS PUBS) are issued by the National Institute of Standards and Technology (NIST) after approval by the Secretary of Commerce pursuant to Section 5131 of the Information Technology Management Reform Act of 1996 (Public Law 104-106) and the Computer Security Act of 1987 (Public Law 100-235).

- Name of Standard.** Advanced Encryption Standard (AES) (FIPS PUB 197).
- Category of Standard.** Computer Security Standard, Cryptography.
- Explanation.** The Advanced Encryption Standard (AES) specifies a FIPS approved cryptographic algorithm that can be used to protect electronic data. The AES algorithm is a symmetric block cipher that can encrypt (encipher) and decrypt (decipher) information. Encryption converts data to an unintelligible form called ciphertext; decrypting the ciphertext converts the data back into its original form, called plaintext.

The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits.

- Approving Authority.** Secretary of Commerce.
- Maintenance Agency.** Department of Commerce, National Institute of Standards and Technology, Information Technology Laboratory (ITL).
- Applicability.** This standard may be used by Federal departments and agencies when an agency determines that sensitive (unclassified) information (as defined in P. L. 100-235) requires cryptographic protection.

Other FIPS-approved cryptographic algorithms may be used in addition to, or in lieu of, this standard. Federal agencies or departments that use cryptographic devices for protecting classified information can use those devices for protecting sensitive (unclassified) information in lieu of this standard.

In addition, this standard may be adopted and used by non-Federal Government organizations. Such use is encouraged when it provides the desired security for commercial and private organizations.

**FIPS PUB 198-1**

---

**FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION**

**The Keyed-Hash Message Authentication Code (HMAC)**


---

CATEGORY: COMPUTER SECURITY    SUBCATEGORY: CRYPTOGRAPHY

---

Information Technology Laboratory  
National Institute of Standards and Technology  
Gaithersburg, MD 20899-8900

July 2008



U.S. Department of Commerce  
*Carlos M. Gutierrez, Secretary*

National Institute of Standards and Technology  
*James M. Turner, Deputy Director*

# Cryptol and Specifications

- Declarative language tailored to cryptography
- Typed functional language with sized vectors

- Bit manipulation

```
ext : {n} (fin n) => [n] -> [n+1]  
ext(x) = x # zero
```

- Safe addition

```
add_safe : {n} (fin n) => ([n], [n]) -> [n+1];  
add_safe(x,y) = ext x + ext y
```



# Case Study: OpenSSL AES



- **Completely automated** once specification and implementation lined up
- Script written primarily by OpenSSL developer
- High end of computational cost
  - Several hours to complete proof
- **Integrated** into fork of OpenSSL repo

## Case Study: Galois ECDSA

- In-house implementation of Elliptic Curve Digital Signature Algorithm (ECDSA) in Java
- Designed for **speed** and **verifiability**
  - Fastest Java implementation we know of
- ~4.5KLLOC of implementation, ~1.5KLLOC proof script
- Around 3.5min to verify on this laptop
- Discovered a **subtle bug**
  - Very similar to the OpenSSL modular reduction one

## The Bug (similar to OpenSSL's)

NISTCurve.java (line 964):

```
d = (z[ 0] & LONG_MASK) + of;  
z[ 0] = (int) d; d >>= 32;  
d = (z[ 1] & LONG_MASK) - of;  
z[ 1] = (int) d; d >>= 32;  
d += (z[ 2] & LONG_MASK);
```



# The Bug (like OpenSSL's)

NISTCurve.java (line 964):

```
d = (d + of) % p;  
z[ 0] = (int) d; d >>= 32;  
d += (z[ 1] & LONG_MASK) - of;  
z[ 1] = (int) d; d >>= 32;  
d += (z[ 2] & LONG_MASK);
```

ABC found bug in 20 seconds.  
Testing found bug after 2 hours  
(8 billion field reductions).

Bug only occurs when this addition overflows.

Previous code guaranteed that  $0 < of < 5$



# Case Study: s2n HMAC

- Amazon's TLS implementation
- Various cryptographic algorithms, including HMAC

$$HMAC(K, text) = H((K_0 \oplus opad) || H((K_0 \oplus ipad) || text))$$

$$HMAC(K, text) = H((K_0 \wedge opad) \# H((K_0 \wedge ipad) \# text))$$

where

$K_0$  = kinit K

ipad = repeat 0x36

opad = repeat 0x5C

```
static int s2n_hmac_init(struct s2n_hmac_state *state, const void *key,
    uint32_t klen)
{
    s2n_hash_algorithm hash_alg = S2N_HASH_NONE;
    if (klen == S2N_HASH_SHA1_DIGEST_SIZE) {
        hash_alg = S2N_HASH_SHA1;
    } else if (klen == S2N_HASH_SHA256_DIGEST_SIZE) {
        hash_alg = S2N_HASH_SHA256;
    } else if (klen == S2N_HASH_SHA512_DIGEST_SIZE) {
        hash_alg = S2N_HASH_SHA512;
    } else {
        return -1;
    }

    for (int i = 0; i < state->block_size; i++) {
        state->xor_pad[i] = 0x36;
    }

    GUARD(s2n_hash_init(&state->inner, hash_alg));
    GUARD(s2n_hash_update(&state->inner, key, klen));
    GUARD(s2n_hash_update(&state->inner, state->xor_pad, state->block_size));

    for (int i = 0; i < state->block_size; i++) {
        state->xor_pad[i] = 0x5C;
    }

    GUARD(s2n_hash_init(&state->outer, hash_alg));
    GUARD(s2n_hash_update(&state->outer, key, klen));
    GUARD(s2n_hash_update(&state->outer, state->xor_pad, state->block_size));

    /* Copy inner just key to inner */
    return s2n_hmac_reset(state);
}

static int s2n_hmac_digest(struct s2n_hmac_state *state, void *out,
    uint32_t size)
{
    for (int i = 0; i < state->block_size; i++) {
        state->xor_pad[i] = 0x36;
    }

    GUARD(s2n_hash_digest(&state->inner, state->digest_pad,
        state->digest_size));
    memory_check(&state->inner, state->digest_size, state->digest_size);
    GUARD(s2n_hash_update(&state->inner, state->digest_pad,
        state->digest_size));

    return s2n_hash_digest(&state->inner, out, size);
}

int s2n_hmac_init(struct s2n_hmac_state *state, s2n_hash_algorithm alg,
    const void *key, uint32_t klen)
{
    s2n_hash_algorithm hash_alg = S2N_HASH_NONE;
    state->currently_in_hash_block = 0;
    state->digest_size = 0;
    state->block_size = 64;
    state->hash_block_size = 64;

    switch (alg) {
        case S2N_HASH_SHA1:
            break;
        case S2N_HASH_SHA256:
            state->block_size = 48;
            /* #12 through ... */
        case S2N_HASH_SHA256:
            hash_alg = S2N_HASH_SHA256;
            state->digest_size = S2N_HASH_SHA256_DIGEST_SIZE;
            break;
        case S2N_HASH_SHA512:
            state->block_size = 48;
            /* #12 through ... */
        case S2N_HASH_SHA512:
            hash_alg = S2N_HASH_SHA512;
            state->digest_size = S2N_HASH_SHA512_DIGEST_SIZE;
            break;
        case S2N_HASH_SHA384:
            hash_alg = S2N_HASH_SHA384;
            state->digest_size = S2N_HASH_SHA384_DIGEST_SIZE;
            state->block_size = 128;
            state->hash_block_size = 128;
            break;
        case S2N_HASH_SHA512:
            hash_alg = S2N_HASH_SHA512;
            state->digest_size = S2N_HASH_SHA512_DIGEST_SIZE;
            state->block_size = 128;
            state->hash_block_size = 128;
            break;
        default:
            return -1;
    }

    return s2n_hmac_init(&state->inner, alg, key, klen);
}

int s2n_hmac_update(struct s2n_hmac_state *state, const void *in, uint32_t size)
{
    /* Keep track of how much of the current hash block is full */
    /* Why the 4294967296 constant in this code? 4294967296 is the
     * highest 32-bit value that is congruent to 0 modulo all of our
     * HMAC block sizes. That is also at least 16x smaller than 2^32, so
     * therefore has no effect on the mathematical result, and no valid
     * record size can cause it to overflow.
     * The value was found with the following python code:
     *
     * x = (2 ** 32) - (2 ** 14)
     * while True:
     *     if x % 40 | x % 48 | x % 64 | x % 128 == 0:
     *         x = x - 1
     *         print x
     *
     * What it does do however is ensure that the mod operation takes a
     * constant number of instruction cycles, regardless of the size of
     * the input. On some platforms, including Intel, the operation can
     * take a smaller number of cycles if the input is "small".
     */
    state->currently_in_hash_block += (uint32_t) size % state->hash_block_size;
    state->currently_in_hash_block %= state->hash_block_size;

    return s2n_hash_update(&state->inner, in, size);
}

int s2n_hmac_digest(struct s2n_hmac_state *state, void *out, uint32_t size)
{
    if (state->alg == S2N_HASH_SHA1 || state->alg == S2N_HASH_SHA256) {
        return s2n_hmac_digest(&state->inner, out, size);
    }

    GUARD(s2n_hash_digest(&state->inner, state->digest_pad,
        state->digest_size));
    GUARD(s2n_hash_update(&state->outer, state->digest_pad,
        state->digest_size));
    GUARD(s2n_hash_update(&state->outer, state->xor_pad, state->block_size));

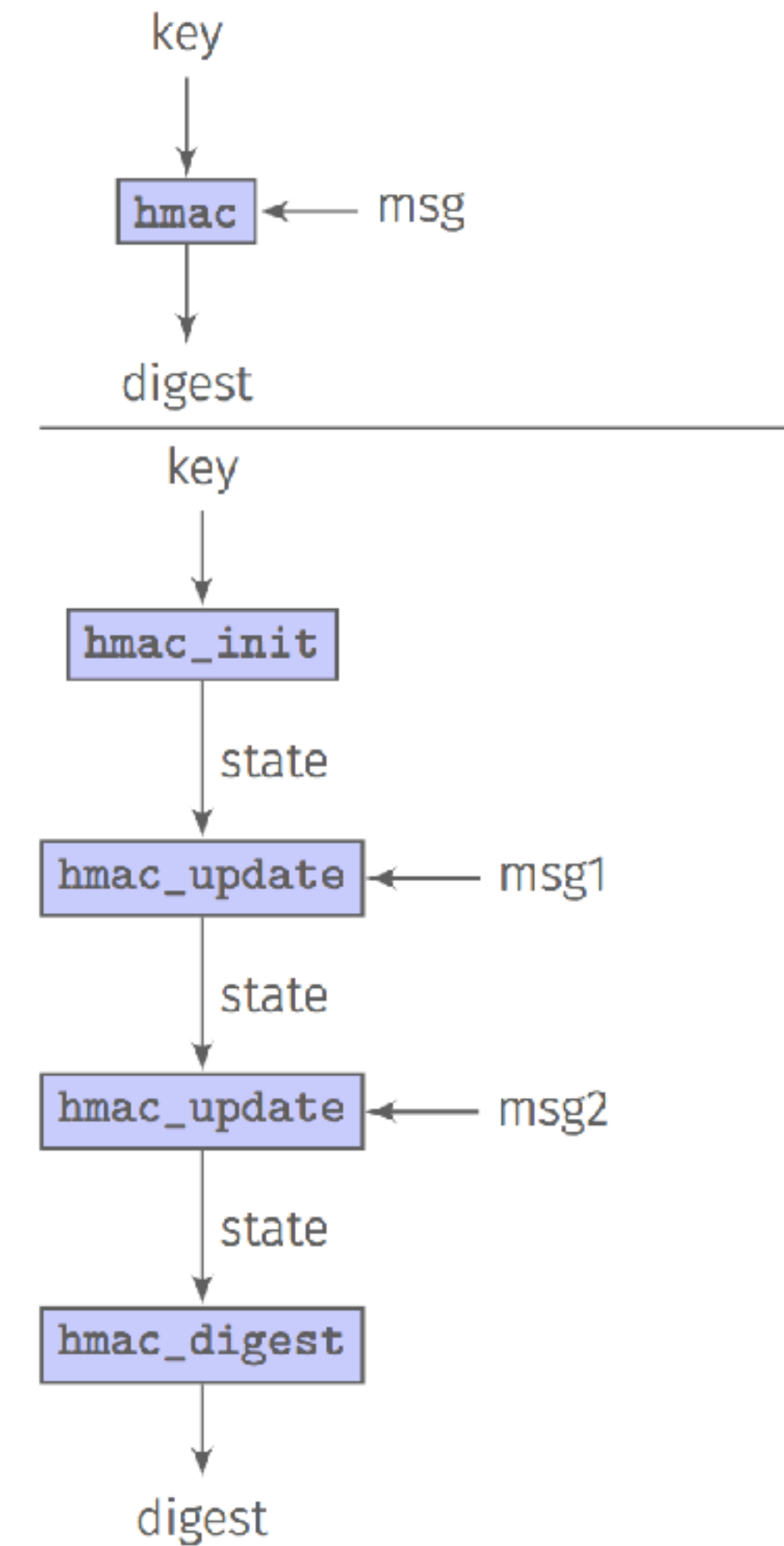
    return s2n_hash_digest(&state->outer, out, size);
}

int s2n_hmac_reset(struct s2n_hmac_state *state)
{
    state->currently_in_hash_block = 0;
    memory_check(&state->inner, state->digest_size, state->digest_size);
    return 0;
}

```

# HMAC Structure

- Specification is a single function
  - Processes whole message at once
- Implementation is incremental
  - Processes message in chunks, as available



# Verification Approach for s2n HMAC

- Transitive verification



- Each stage **automatically** proved
- Integrated into Travis CI system
  - Code changes **re-verified** on every commit

# s2n

s2n is a C99 implementation of the TLS/SSL protocols that is designed to be simple, small, fast, and with security as a priority. It is released and licensed under the Apache Software License 2.0.

build **passing** license **Apache License 2.0** language **C99** forks **210** stars **2k**

## Using s2n

The s2n I/O APIs are designed to be intuitive to developers familiar with the widely-used POSIX I/O APIs, and s2n supports blocking, non-blocking, and full-duplex I/O. Additionally there are no locks or mutexes within s2n.

```
/* Create a server mode connection handle */
struct s2n_connection *conn = s2n_connection_new(S2N_SERVER);
if (conn == NULL) {
    ... error ...
}

/* Associate a connection with a file descriptor */
if (s2n_connection_set_fd(conn, fd) < 0) {
    ... error ...
}
```

# The Future of SAW's Approach

- Not just cryptographic code. Some early success:
  - Serialization and deserialization
  - DSP
- Fewer constraints. Have design concepts for:
  - Unbounded loops
  - Non-fixed heaps
  - Easier compositional reasoning

# Wrapping Up

- Software is a **mathematical artifact**
- Conclusive **proofs** about its behavior are possible
- SAW partly **automates** this in an open source tool
- Particularly effective for **cryptographic** code
- Broader applications likely in the future

# Acknowledgements

Aaron Tomb, Adam Foltzer, Adam Wick, Andrey Chudnov, Andy Gill, Benjamin Barenblat, Ben Jones, Brian Huffman, Brian Ledger, David Lazar, Dylan McNamee, Edward Yang, Eric Mertens, Fergus Henderson, Iavor Diatchki, Jeff Lewis, Jim Teisher, Joe Hendrix, Joe Hurd, Joe Kiniry, Joel Stanley, Joey Dodds, John Launchbury, John Matthews, Jonathan Daugherty, Kenneth Foner, Kyle Carter, Ledah Casburn, Lee Pike, Levent Erkök, Magnus Carlsson, Mark Shields, Mark Tullsen, Matt Sottile, Nathan Collins, Philip Weaver, Robert Dockins, Sally Browning, Sam Anklesaria, Sigbjørn Finne, Thomas Nordin, Trevor Elliott, and Tristan Ravitch.



# Resources

- Contact me
  - Aaron Tomb <[atomb@galois.com](mailto:atomb@galois.com)>
- SAW is freely available and open source
  - <http://saw.galois.com>
  - <https://github.com/GaloisInc/saw-script>
- Cryptol is freely available and open source
  - <http://cryptol.net>
  - <https://github.com/GaloisInc/cryptol>
- HMAC Verification
  - <https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/>