Proving the Correctness of Amazon's s2n TLS Library

Aaron Tomb <atomb@galois.com>
Galois, Inc.

May 11, 2018
International Cryptographic Modules Conference

The s2n TLS Library

- Amazon's open source implementation of TLS
  - *"designed to be simple, small, fast, and with security as a priority"*
- We want a high degree of assurance in its correctness
  - Widely used libraries have had serious bugs in the past
- And to retain that assurance as it evolves
  - Many implementation choices, so code changes even when the specification doesn't
- This talk: using automated reasoning check correctness
  - Integrated with Travis CI to re-check every change

Cryptographic Assurance via Testing

- Typical approach: run on test vectors
  - From NIST or other authority
  - One part of FIPS certification
- But bugs occur anyway!
- Some undefined behavior: Heartbleed
  - Buffer over-read in OpenSSL
  - A generic misuse of language
- Some incorrect behavior: OpenSSL ECC modular reduction
  - Incorrect modular reduction
  - Requires knowing correct results

Practical realisation and elimination of an ECC-related software bug attack*

B. B. Brumley[1], M. Barbosa[2], D. Page[3], and F. Vercauteren[4]

[1] Department of Information and Computer Science, Aalto University School of Science, P.O. Box 15400, FI-00076 Aalto, Finland.
billy.brumley@aalto.fi
[2] HASLab/INESC TEC Universidade do Minho, Braga, Portugal.
mbb@di.uminho.pt
[3] Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK.
page@cs.bris.ac.uk
[4] Department of Electrical Engineering, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium.
fvercaut@esat.kuleuven.ac.be

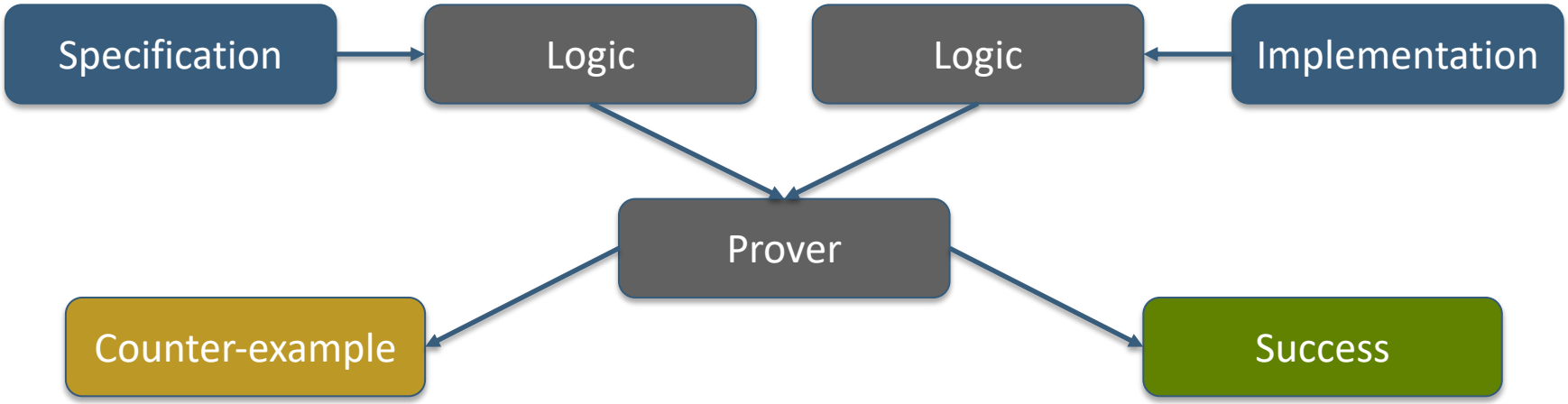
Abstract. We analyse and exploit implementation features in OpenSSL version 0.9.8*g* which permit an attack against ECDH-based functionality. The attack, although more general, can recover the entire (static) private key from an associated SSL server via 633 adaptive queries when the NIST curve P-256 is used. One can view it as a software-oriented analogue of the bug attack concept due to Biham et al. and, consequently, as the first bug attack to be successfully applied against a real-world system. In addition to the attack and a posteriori countermeasures, we show

Generic vs. Application-Specific Bugs

- Heartbleed is a generic bug:
  - Undefined behavior incorrect regardless of the specification
  - Most code doesn't have an unambiguous specification
  - Most static analysis tools focus here, for generality
- ECC bug is application-specific:
  - Well-defined, but doesn't match specification
  - Crypto code often does have a specification!
  - This talk focuses here
- If we have an executable specification, we can test against it!
  - With many more tests that a static set of test vectors
- But for realistic programs, infinite inputs (or effectively so)

Exhaustive Testing via Automated Reasoning

- But exhaustive testing is possible (given a full specification)
  - Enabled by automated reasoning tools: SAT and SMT
  - Specification and production implementation can be translated to logic
  - Automated provers can show that they produce the same results for all possible inputs
- Specification must be machine-readable



Cryptol and SAW

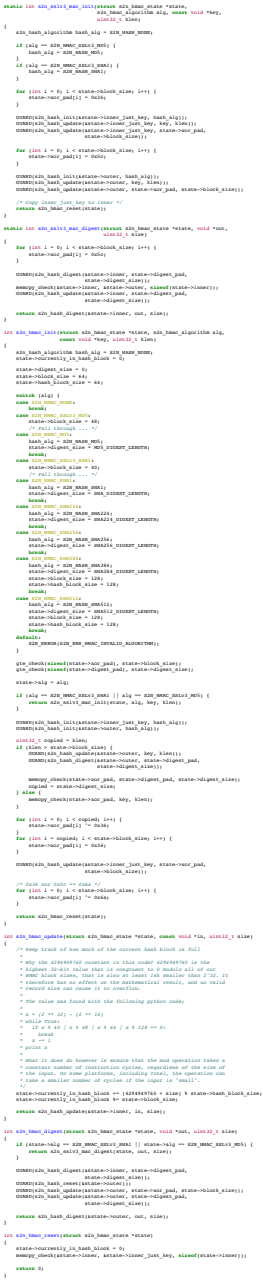
- Open source tools for software analysis
  - Especially cryptographic software verification
- Cryptol allows us to write concise, unambiguous specifications
  - Specifications exist for many common algorithms
  - They closely resemble specification documents
- The Software Analysis Workbench (SAW) can compare Cryptol specifications and code
  - Extracts models from programs
  - Transforms and proves properties of models
  - Supports languages compiled to JVM and LLVM (with more in development)
  - Uses SAT & SMT in the background

HMAC

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

```
HMAC(K, text) = H((K0 ^ opad) # H((K0 ^ ipad) # text))
  where
    K0 = kinit K
    ipad = repeat 0x36
    opad = repeat 0x5C
```

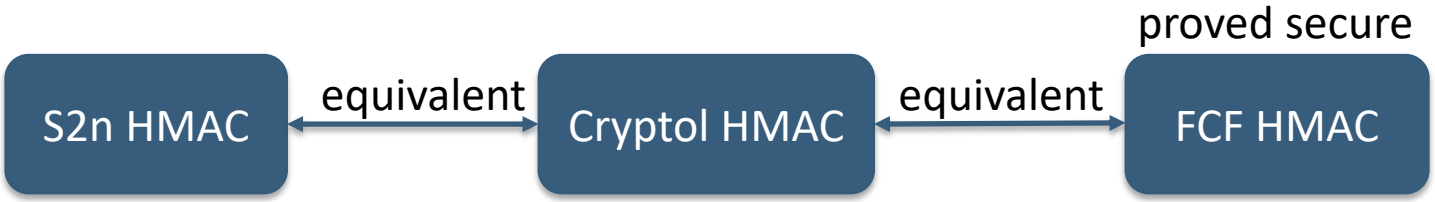
- Keyed-Hash Message Authentication Code
  - Specified in FIPS 198-1
- NIST document and Cryptol match closely
  - But Cryptol is less ambiguous
- Specification is much simpler!
  - 5 LOC in Cryptol vs. ~200 LOC in C



```
static int s2n_sslv3_mac_init(struct s2n_hmac_state *state,
                              s2n_hmac_algorithm alg, const void *key,
                              uint32_t klen)
{
    s2n_hash_algorithm hash_alg = S2N_HASH_NONE;

    if (alg == S2N_HMAC_SSLV3_MD5) {
        hash_alg = S2N_HASH_MD5;
    }
    if (alg == S2N_HMAC_SSLV3_SHA1) {
        hash_alg = S2N_HASH_SHA1;
    }

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

    GUARD(s2n_hash_init(&state->inner_just_key, hash_alg));
    GUARD(s2n_hash_update(&state->inner_just_key, key, klen));
    GUARD(s2n_hash_update(&state->inner_just_key, 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_sslv3_mac_digest(struct s2n_hmac_state *state, void *out,
                                uint32_t size);
{
    for (int i = 0; i < state->block_size; i++) {
        state->xor_pad[i] = 0x5c;
    }

    GUARD(s2n_hash_digest(&state->inner, state->digest_pad,
                          state->digest_size));
    memcpy_check(&state->inner, &state->outer, sizeof(state->inner));
    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_hmac_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_HMAC_NONE:
        break;
    case S2N_HMAC_SSLV3_MD5:
        state->block_size = 48;
        /* Fall through ... */
    case S2N_HMAC_MD5:
        hash_alg = S2N_HASH_MD5;
        state->digest_size = MD5_DIGEST_LENGTH;
        break;
    case S2N_HMAC_SSLV3_SHA1:
        state->block_size = 40;
        /* Fall through ... */
    case S2N_HMAC_SHA1:
        hash_alg = S2N_HASH_SHA1;
        state->digest_size = SHA_DIGEST_LENGTH;
        break;
    case S2N_HMAC_SHA224:
        hash_alg = S2N_HASH_SHA224;
        state->digest_size = SHA224_DIGEST_LENGTH;
        break;
    case S2N_HMAC_SHA256:
        hash_alg = S2N_HASH_SHA256;
        state->digest_size = SHA256_DIGEST_LENGTH;
        break;
    case S2N_HMAC_SHA384:
        hash_alg = S2N_HASH_SHA384;
        state->digest_size = SHA384_DIGEST_LENGTH;
        state->block_size = 128;
        state->hash_block_size = 128;
        break;
    case S2N_HMAC_SHA512:
        hash_alg = S2N_HASH_SHA512;
        state->digest_size = SHA512_DIGEST_LENGTH;
        state->block_size = 128;
        state->hash_block_size = 128;
        break;
    default:
        S2N_ERROR(S2N_ERR_HMAC_INVALID_ALGORITHM);
    }

    gte_check(sizeof(state->xor_pad), state->block_size);
    gte_check(sizeof(state->digest_pad), state->digest_size);

    state->alg = alg;

    if (alg == S2N_HMAC_SSLV3_SHA1 || alg == S2N_HMAC_SSLV3_MD5) {
        return s2n_sslv3_mac_init(state, alg, key, klen);
    }

    GUARD(s2n_hash_init(&state->inner_just_key, hash_alg));
    GUARD(s2n_hash_init(&state->outer, hash_alg));

    uint32_t copied = klen;
    if (klen > state->block_size) {
        GUARD(s2n_hash_update(&state->outer, key, klen));
        GUARD(s2n_hash_digest(&state->outer, state->digest_pad,
                              state->digest_size));

        memcpy_check(&state->xor_pad, state->digest_pad, state->digest_size);
        copied = state->digest_size;
    } else {
        memcpy_check(&state->xor_pad, key, klen);
    }

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

    GUARD(s2n_hash_update(&state->inner_just_key, state->xor_pad,
                          state->block_size));

    /* 0x36 xor 0x5c == 0x6a */
    for (int i = 0; i < state->block_size; i++) {
        state->xor_pad[i] ^= 0x6a;
    }

    return s2n_hmac_reset(state);
}

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 4294967960 constant in this code? 4294967960 is the
     * highest 32-bit value that is congruent to 0 modulo all of our
     * HMAC block sizes, that is also at least 16k smaller than 2^32. It
     * therefore has no effect on the mathematical result, and so 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:
     *     break
     *   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 += (4294967960 + size) % state->hash_block_size;
    state->currently_in_hash_block %= state->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_HMAC_SSLV3_SHA1 || state->alg == S2N_HMAC_SSLV3_MD5) {
        return s2n_sslv3_mac_digest(state, out, size);
    }

    GUARD(s2n_hash_digest(&state->inner, state->digest_pad,
                          state->digest_size));
    GUARD(s2n_hash_reset(&state->outer));
    GUARD(s2n_hash_update(&state->outer, state->xor_pad, state->block_size));
    GUARD(s2n_hash_update(&state->outer, state->digest_pad,
                          state->digest_size));

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

int s2n_hmac_reset(struct s2n_hmac_state *state)
{
    state->currently_in_hash_block = 0;
    memcpy_check(&state->inner, &state->inner_just_key, sizeof(state->inner));

    return 0;
}
```

HMAC: Cryptographic Strength

- HMAC proved secure (by hand) (Bellare, et al., 1996, 2006)
- Later, machine-checked using the Foundational Cryptography Framework (FCF) (Beringer et al., 2015)
- We connected FCF to Cryptol
- Result: the s2n HMAC implementation:
  - is equivalent to the NIST specification, and
  - produces output indistinguishable from random.

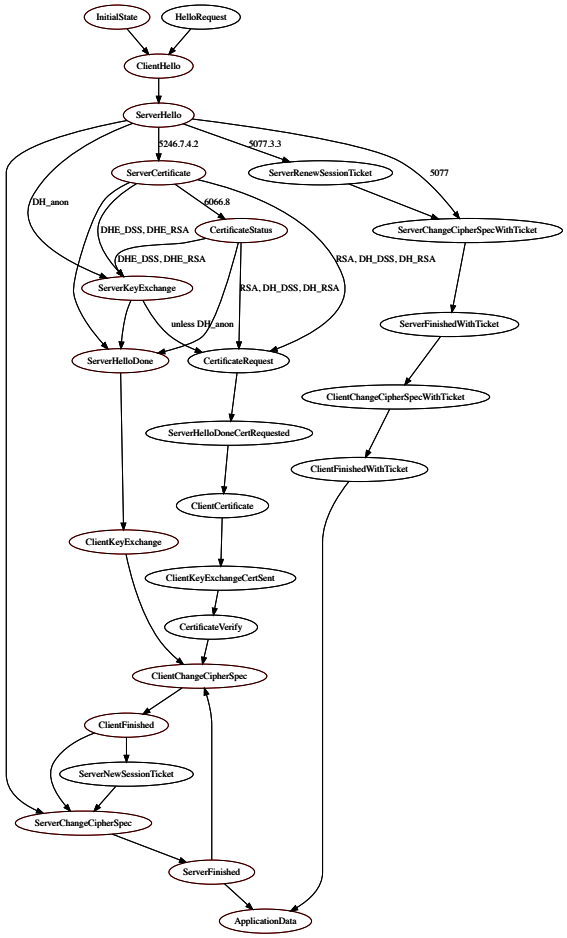


DRBG

- Deterministic Random Bit Generator
  - Specified in NIST SP 800-90A
  - Instantiated in CTR mode with AES-128
- Specification is similar in structure to implementation
  - 81 lines of Cryptol vs. 187 lines of C
  - Closer than HMAC because DRBG specified imperatively
  - Specification is shorter…
  - …and could be used to verify multiple implementations!
- Security of the specification is an open question
  - Though machine-checked proof exists for HMAC_DRBG

TLS Handshake

- Coordinates cipher choice, key exchange
  - Specified in RFC 5246
  - Famous bugs here, e.g., SKIP-TLS
- Specification is very different from code
  - s2n implements a subset of the RFC
  - RFC spec: 200 lines
  - Subset spec: 242 lines
  - C code: 463 lines
  - Tables are most of spec and C
- Discovered a bug through verification
  - Not security-critical, but incorrect



Continuous Integration

- All of this re-runs on every commit, using Travis CI
- Runs faster than the concrete tests: 3-15 minutes per proof

Build Jobs

✓	# 1730.1	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.1.0 BUILD_S2N=true TESTS=integrati···	18 min 57 sec
✓	# 1730.2	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.0.2 BUILD_S2N=true TESTS=integrati···	30 min 23 sec
✓	# 1730.3	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.0.2-fips BUILD_S2N=true TESTS=inte···	34 min 13 sec
✓	# 1730.4	Xcode: xcode8 C	S2N_LIBCRYPTO=libressl BUILD_S2N=true TESTS=integration GC···	22 min 9 sec
✓	# 1730.5	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.1.0 OPENSSL_ia32cap="~0x2000002···	18 min 32 sec
✓	# 1730.6	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.1.0 LATEST_CLANG=true TESTS=fuzz ···	18 min 53 sec
✓	# 1730.7	Xcode: xcode8 C	S2N_LIBCRYPTO=openssl-1.0.2-fips LATEST_CLANG=true TESTS=···	31 min 16 sec
✓	# 1730.8	Xcode: xcode8 C	TESTS=sidewinder	36 min 43 sec
✓	# 1730.9	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=md5 SAW=true GCC6_REQUI···	14 min 37 sec
✓	# 1730.10	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha1 SAW=true GCC6_REQU···	13 min 15 sec
✓	# 1730.11	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha224 SAW=true GCC6_RE···	12 min 58 sec
✓	# 1730.12	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha256 SAW=true GCC6_RE···	11 min 44 sec
✓	# 1730.13	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha384 SAW=true GCC6_RE···	14 min 19 sec
✓	# 1730.14	Xcode: xcode8 C	TESTS=sawHMAC SAW_HMAC_TEST=sha512 SAW=true GCC6_RE···	13 min 58 sec
✓	# 1730.15	Xcode: xcode8 C	TESTS=tls SAW=true GCC6_REQUIRED=false	2 min 33 sec
✓	# 1730.16	Xcode: xcode8 C	TESTS=sawHMACFailure SAW=true	3 min 10 sec

Takeaway

- Given a specification, exhaustive testing is often possible
  - Especially for cryptographic code
  - Enabled by SAT and SMT
- We have done this for parts of the open source s2n TLS library
  - HMAC, DRBG, TLS Handshake Protocol
  - More likely in the future
- Re-checked on every commit, using Travis CI
  - Implementations change frequently, even when specifications are stable

Acknowledgements

Contributors include **Aaron Tomb**, Adam Foltzer, Adam Wick, **Andrey Chudnov**, Andy Gill, Benjamin Barenblat, Ben Jones, Brian Huffman, Brian Ledger, David Lazar, Dylan McNamee, **Eddy Westbrook**, Edward Yang, **Eric Mertens**, **Eric Mullen**, 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, **Stephen Magill**, Thomas Nordin, Trevor Elliott, and Tristan Ravitch.

Resources

- Contact me:
  - Aaron Tomb <atomb@galois.com>
- SAW is open source, BSD3 licensed, on GitHub:
  - http://saw.galois.com
  - https://github.com/GaloisInc/saw-script
- Cryptol open source, BSD3 licensed, on GitHub:
  - http://cryptol.net
  - https://github.com/GaloisInc/cryptol
- HMAC verification blog post:
  - https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/