Implementing Navigation Message Authentication on SBAS

SPENCER PAUL'S REU

Stanford University GPS Lab

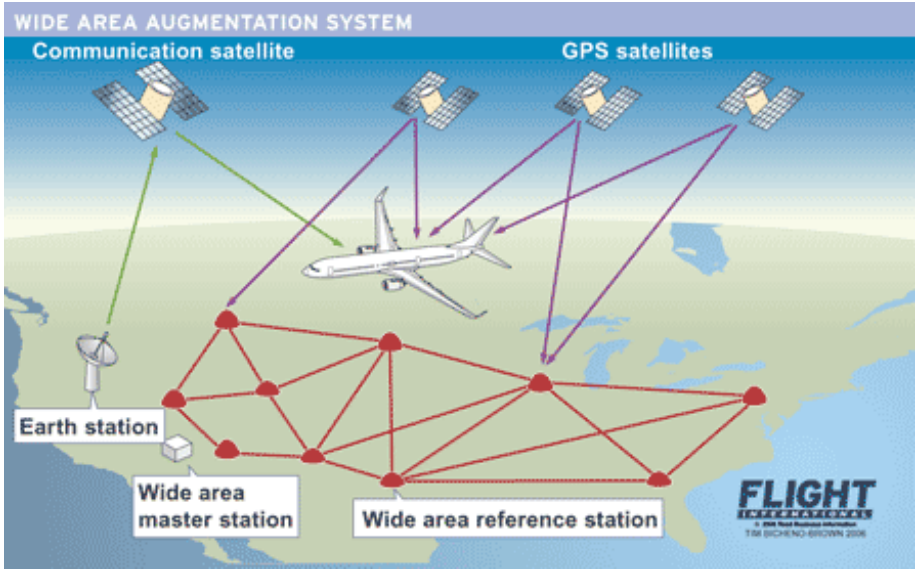
6/1/21

Background Information

SBAS, CRYPTOGRAPHY

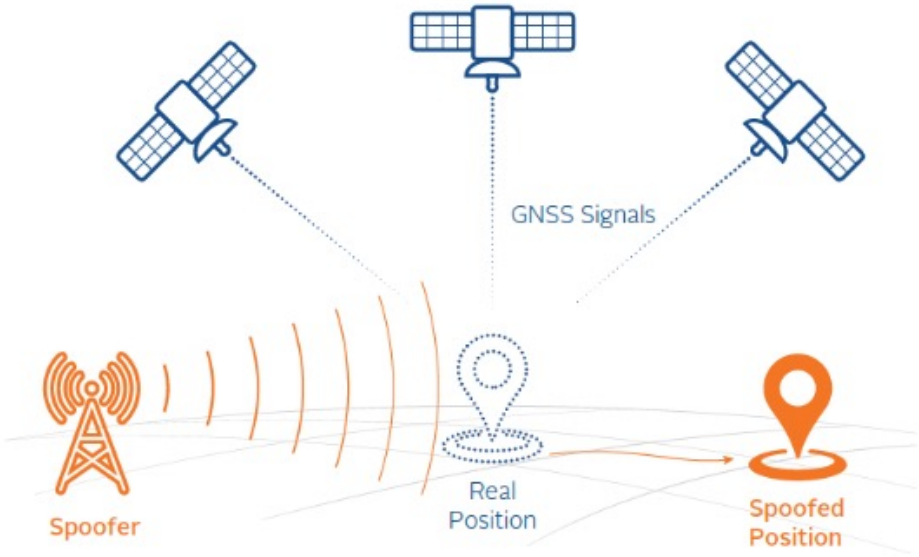
Satellite-based Augmentation System (SBAS)

- SBAS uses a collection of listening stations at **known** position to provide corrections
- Provides augmentation information for corrections of satellite position errors, clock/time errors and errors induced by delay of the signal while crossing the ionosphere.
- increases the accuracy with position errors below 1 meter



What is Spoofing?

- malicious actors want to trick (spoof) GNSS users by manipulating their location
- Areas this could potentially occur -- smartphone to produce false Uber charges, mislead a passenger jet or autonomous vehicle



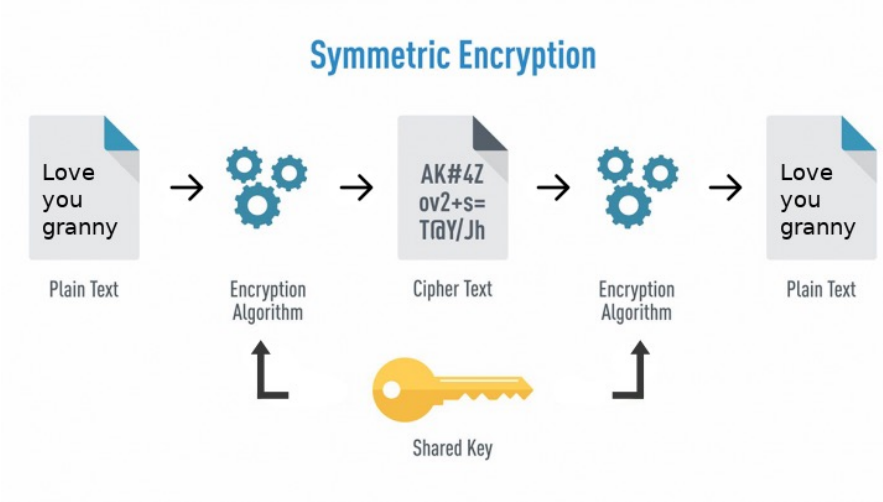
Authentication is used to prevent spoofing

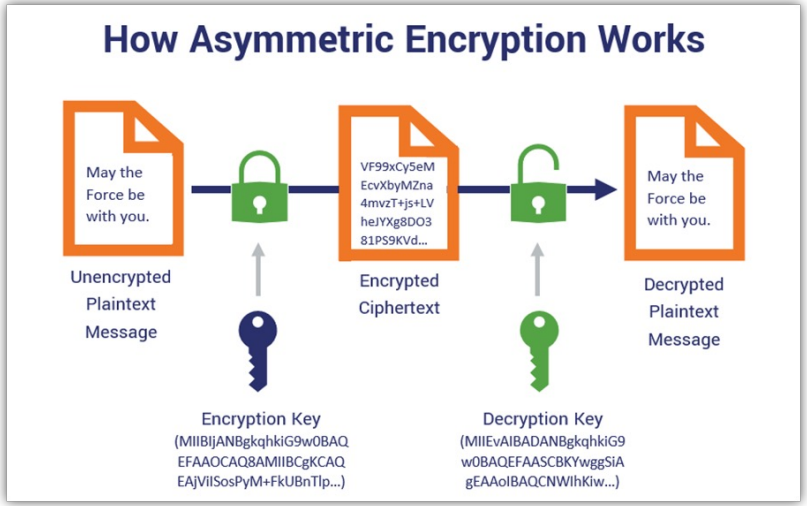
- Authentication is the process of determining if a claim is true
  - i.e gmail authenticates you are yourself when you log into your email
  - Stanford makes you scan an ID card to prove your identity when entering buildings

- This is distinct from encryption which encodes information that is not comprehensible without a specific key to decode it

- My work focused on implementing navigation message authentication (NMA) on SBAS to hinder spoofing

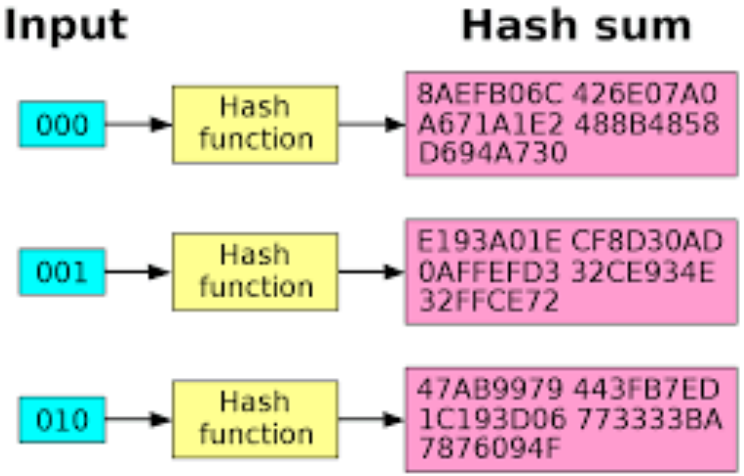
Asymmetric v Symmetric Encryption





Hash Functions

- Deterministic – each input has the same output every time
- One Way – it is very hard to determine input from hash
- Collision Free – no two inputs will have the same hash sum



Our Scheme

TESLA-ECDSA

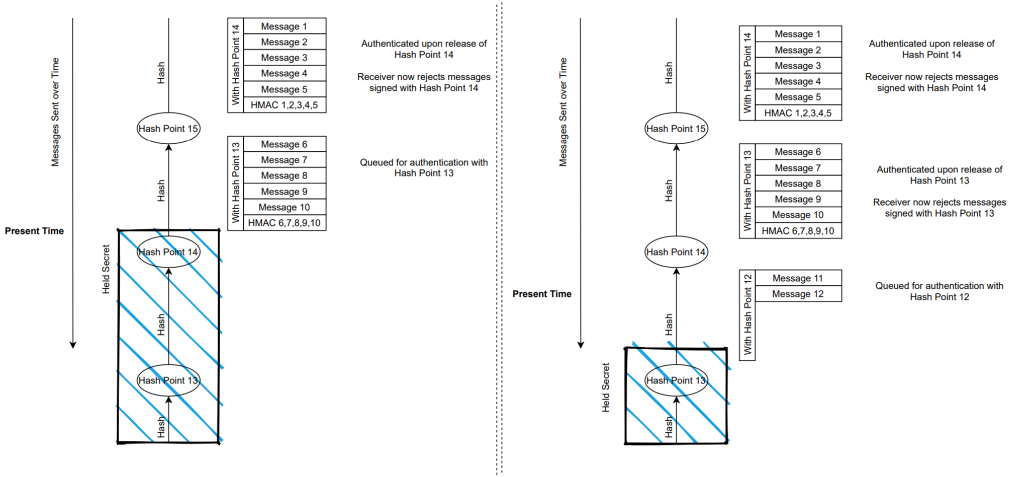
Tesla-ECDSA Scheme

- TESLA must be used in tandem with ECDSA
  - TESLA authenticates the SBAS messages, and ECDSA authenticates SBAS's TESLA use as periodic maintenance.
  - We modify the SBAS message schedule by appending an extra message after every 5 messages that is used for authentication

- Requirements for design
  - [Jason help]

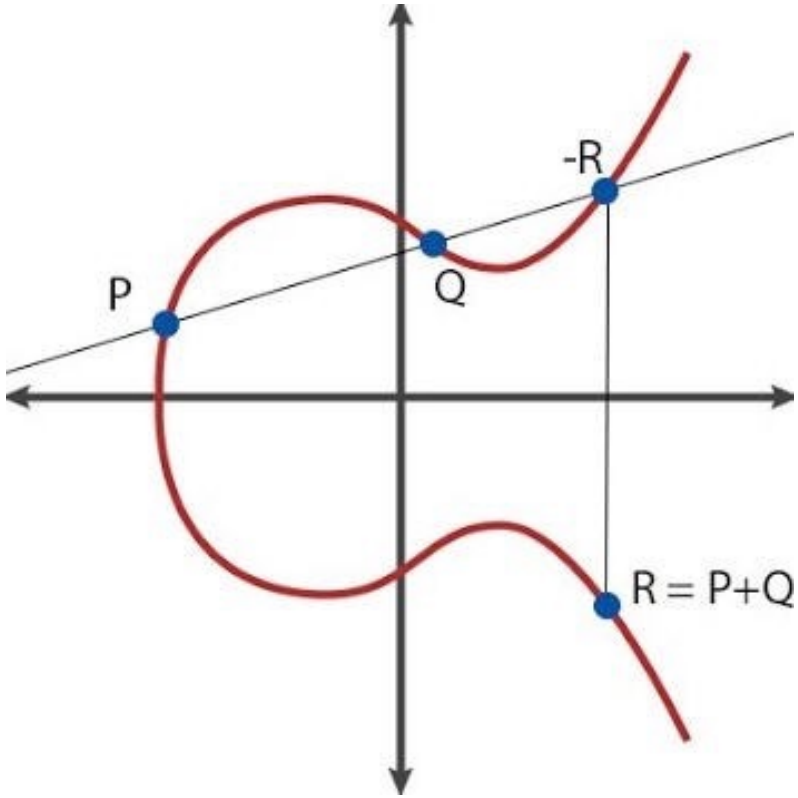
TESLA Portion of Scheme

- Very simple – only requires one cryptographic hash function (h(point))
- Works by generating a chain of hashes p1 = h (p2), p2 = h(p3), p3 = h(p4) …..
  - p4 → p3 → p2 → p1



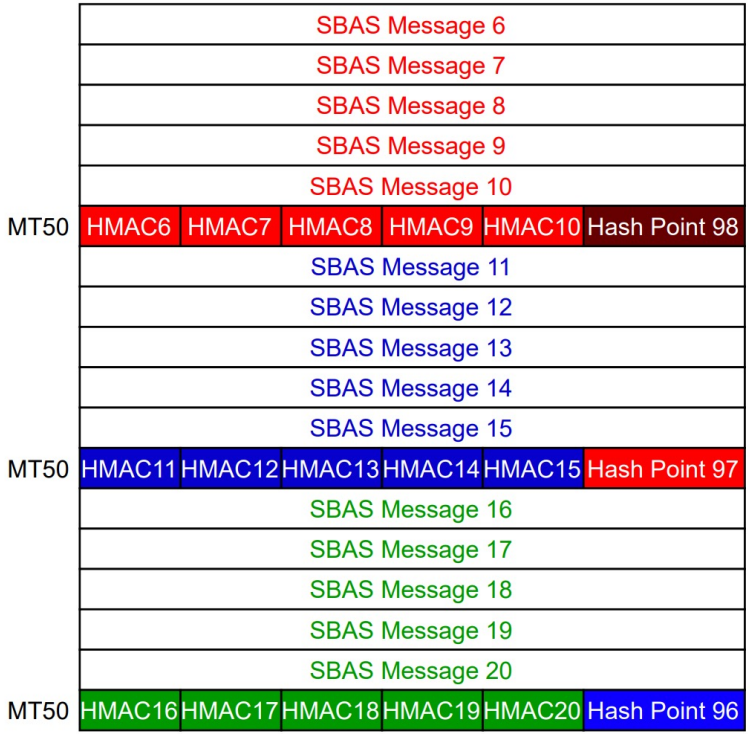
ECDSA

- ECDSA stands for Elliptic Curve Digital Signing algorithm and is used to create a digital signature of data
- Used to verify the first hash point that is released
- This is also asymmetric and relies of the use of curves and mathematical equations to generate a public key that can verify a signature.



Zooming in on the actual messages

- MT50 – message containing authentication data
- HMAC – message authentication code for each message
  - key = xor(hp97, time)
  - HMAC6 = h(message, key)
- 6-11 second delay before we can authenticate messages

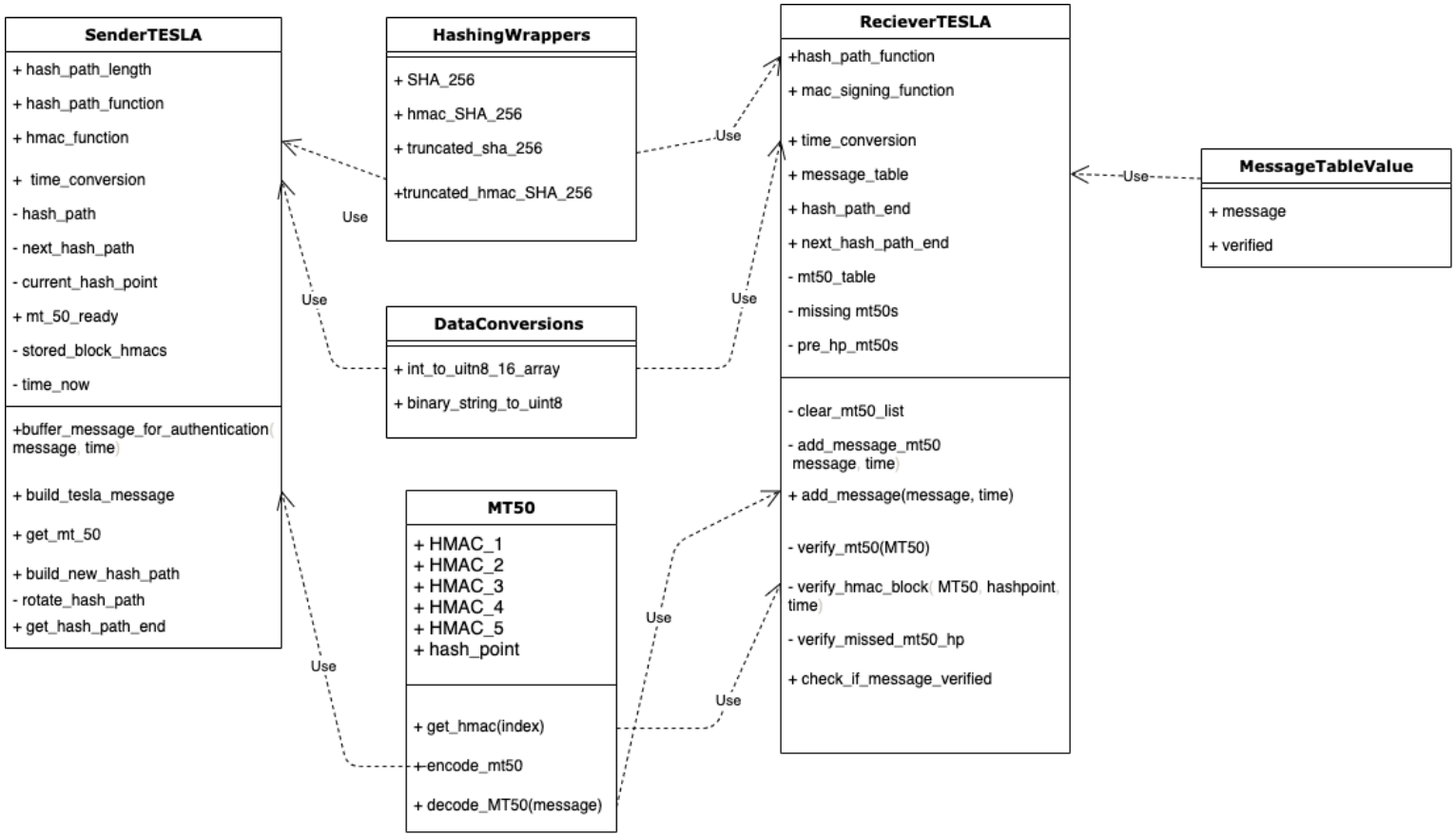


Implementing the Scheme

Programing authentication into MAAST

- MAAST is a simulation on MATLAB of SBAS
- We implemented a full-stack SBAS simulation of our design into MAAST
- MAAST provides Monte-Carlo simulation results to evaluate how our design performs under message loss over the WAAS coverage area
- Our KPIs: time to first authenticated fix, time to authentication per message, and robustness to message loss

Class diagram of what I was coding this quarter



Edge cases in simulation added a lot of complexity

```
methods (Test)

    function test_key_chain_construction(testCase)...

    function test_M50_creation(testCase)...

    function test_next_keychain(testCase)...

    function test_sender_and_reciever(testCase)...

    function test_manipulated_message(testCase)...

    function test_manipulated_mt50(testCase)...

    function test_missing_mt50(testCase)...

    function test_swapped_mt50(testCase)...

    function test_swapped_messages(testCase)...

    function test_sender_and_reciever_keychain_rotation(testCase)...

    function test_missing_message(testCase)...

    function test_many_rotations(testCase)...

    function test_long_string_missed_messages(testCase)...

    function test_constructor_args(testCase)...

    function test_1_arg_constructor(testCase)...
```

Adding missed message functionality

```
function verify_missed_mt50_hp(obj)
    largest_verified_key = max(cell2mat(keys(obj.mt50_table)));
    for i = 1:length(obj.missing_mt50)
        if isKey(obj.mt50_table, obj.missing_mt50{i} - 1) && ...
                largest_verified_key > obj.missing_mt50{i}

            count = largest_verified_key;
            new_hash = obj.mt50_table(largest_verified_key);
            new_hash = new_hash{1}.hash_point;

            while count ~= obj.missing_mt50{i}
                new_hash = obj.key_chain_function(new_hash);
                count = count - 1;
            end

            if obj.missing_mt50{i} - 1 ~= 0
                mt50 = obj.mt50_table(obj.missing_mt50{i} - 1);
                m = mt50{1};
                t = mt50{2};

                obj.verify_hmac_block(m, new_hash, t);
            end
        end
    end

    obj.missing_mt50 = {};
end
```

Hash path rotation

```
function buffer_message_for_authentication(obj, message, time)
    message = DataConversions.binary_string_to_uint8(message);

    % keeping track of curret hmac index and total messages
    obj.message_count = obj.message_count + 1;
    obj.hmac_num = obj.hmac_num + 1;
    % once we reach end of key chain - rotate the hash path
    if obj.key_chain_length - obj.current_block <= 0
        % store last element of key chain for next mt50
        obj.stored_block{6} = obj.key_chain{1};
        obj.rotate_hash_path();
        obj.current_block = 1;
    end
```

Coding Practices

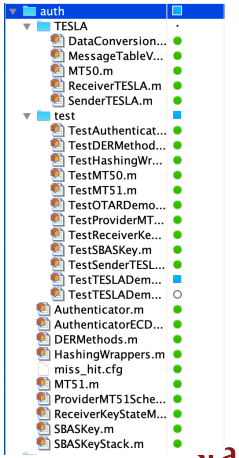
Unit Testing

- Unit Testing was critical to prevent nasty bugs in the future when we integrated code into MAAST
- Every line of code had to be unit tested to pass our build check
- We had ~1500 lines of code dedicated to unit testing ~1500 lines of actual code

matlab unittest coverage report

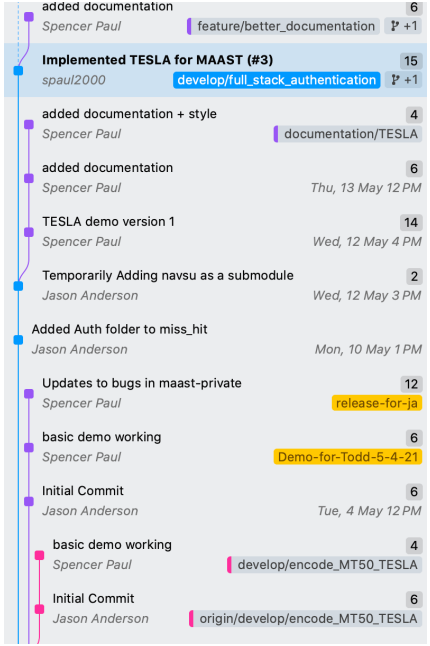
File	Coverage	Lines	Branches		Missing
All files	1%	1%	NaN%	❌	
Authenticator	100%	100%	NaN%	✅	
AuthenticatorECDSA	100%	100%	NaN%	✅	
HashingWrappers	100%	100%	NaN%	✅	
assert_matlab_checkcode	0%	0%	NaN%	❌	4-30
execute_matlab_tests_into_xml	0%	0%	NaN%	❌	13-32
TestAuthenticatorECDSA	100%	100%	NaN%	✅	
TestHashingWrappers	100%	100%	NaN%	✅	

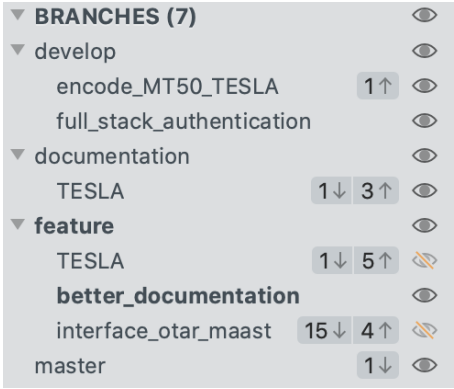
Minimum allowed coverage is `100%`



Using GitHub for organized version control

- Helped Jason and myself collaborate effectively throughout the quarter
- Allowed us to keep a linear version history
- Prevents merge conflicts and allowed us to see exactly where our code diverged





Other Coding Checks

- Mandatory code review for PR approval

- Style Checks using Miss Hit