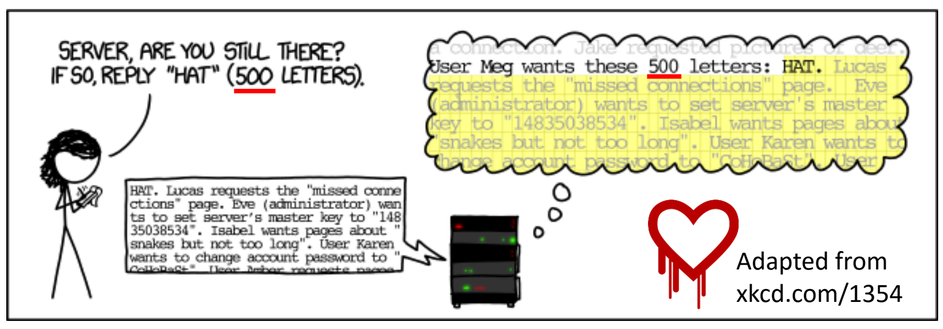
Server-side Verification of Client Behavior in Cryptographic Protocols

Andrew Chi, Robert Cochran, Marie Nesfield, Michael K. Reiter, Cynthia Sturton
University of North Carolina *{achi,nesfield}@cs.unc.edu*

Project Silver

Goals

Exploits of client-server protocols often involve modifying clients to behave in ways that untampered clients could not.



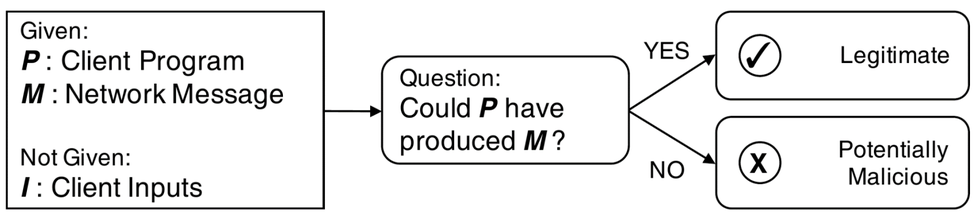
Heartbleed. *OpenSSL bug that affected half a million web servers in 2014. A modified client sends a TLS Heartbeat containing an invalid (but encrypted) length field.*

Current Practice

- *Attempt to harden servers.* Even with extensive review, widely deployed servers have codebases too large to even guarantee perfect input validation (generally considered an "easy" fix).
- *Anomaly detection.* Statistical IDS tools detect egregious client misbehavior, but are vulnerable to mimicry attacks.

Our Innovation

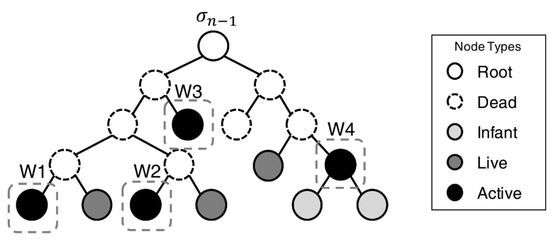
We build a verifier that checks whether client behavior is consistent with the exact client program; we therefore discover client exploit attempts even prior to a vulnerability's disclosure.



Approach

The verifier monitors each client message as it is delivered to the server, and uses symbolic execution to solve for inputs that could have driven the client software to send the observed sequence of messages. To handle real-world cryptographic clients, two core innovations are required:

1. Parallelization: Explore multiple candidate execution paths concurrently using thread-level parallelism.

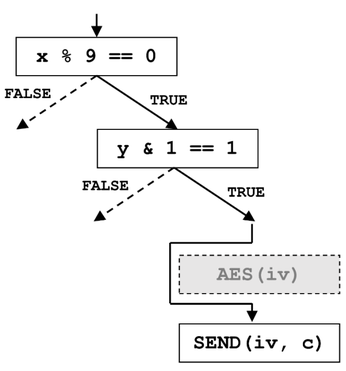


Execution tree explored in parallel by worker threads W1 through W4.

2. Multipass: Cryptographic functions on unknown (symbolic) data are prohibitively costly. Defer them until their inputs can be deduced, by running multiple passes of symbolic execution.

```
void Client(int x, int y, int iv) {
    int p = x * y;
    if (x % 9 == 0) {
        if (y & 1 == 1) {
            int s = AES(iv);
            int c = p ^ s;
            SEND(iv, c);
        }
    }
}
```

Example client code



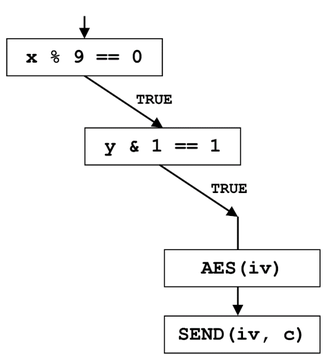

```
Initial State:
x = ?, y = ?, iv = ?

Learned Constraints:
p = x * y
x % 9 = 0
y & 1 = 1
c = p ^ s          } σ.cons

iv = 0x1234        } nd.saved
c = 0x9DAC

Message to Verify:
msg0 = 0x12349DAC
```

Pass one. AES(iv) *is deferred because the verifier cannot deduce* iv = 0x1234 *until reconciliation with the observed message at* SEND(iv, c).

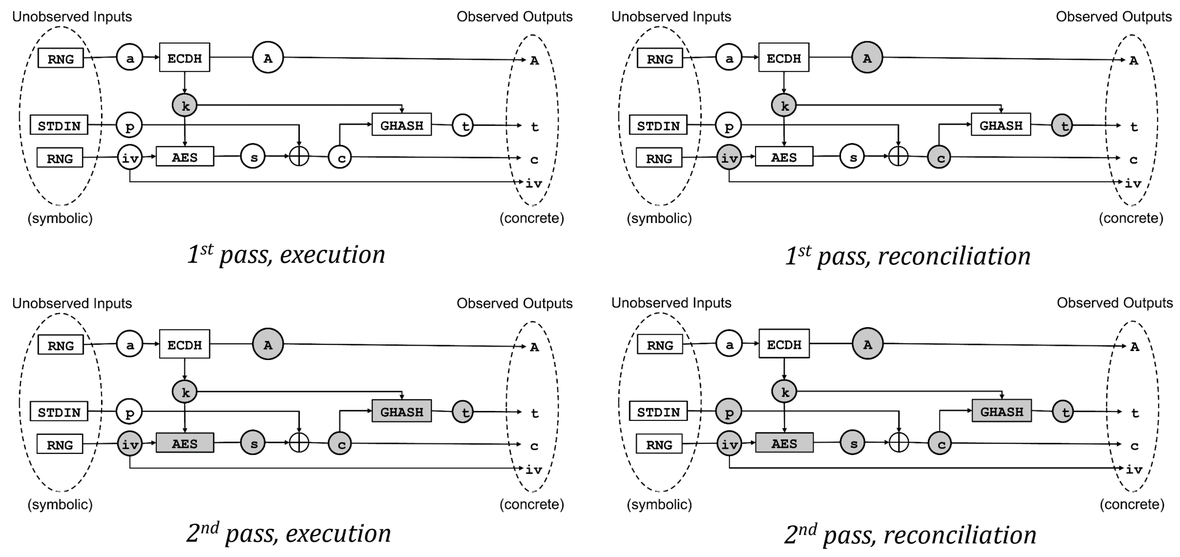



```
Initial State:
x = ?, y = ?
iv = 0x1234, c = 0x9DAC

Learned Constraints:
p = x * y
x % 9 = 0
y & 1 = 1
c = p ^ s          } σ.cons
iv = 0x1234        } ≡
c = 0x9DAC         } nd.saved
s = 0x2343
p = 0xBEEF

Message to Verify:
msg0 = 0x12349DAC
```

Pass two. AES(iv) *executes concretely. Terminates when constraint set* σ.cons *is logically equivalent to its previous version* nd.saved *(i.e., fixed point).*



1st pass, execution *1st pass, reconciliation*

2nd pass, execution *2nd pass, reconciliation*

Multipass algorithm on a TLS client implementing an abstracted subset of AES-GCM. *Rectangular blocks are prohibitive functions; circles are variables. Shaded nodes are concrete values or functions executed with concrete inputs. Unshaded nodes are symbolic values or skipped functions. The symmetric key, k, is communicated by the server to the verifier.*

Implementation

The verifier is built upon KLEE, a symbolic execution engine that operates on LLVM bitcode. Our enhancements comprise:

1. "Client Verification" mode (previous work)
   a. Special handling of `SEND/RECV` network events
   b. Reconcile observed messages against constraints
2. Parallelization of KLEE
   a. Multiple symbolic state "searchers"
   b. Thread-safe symbolic memory management
   c. Thread-safe constraint caching and SAT solving
3. Multipass symbolic execution
   a. API for specifying prohibitive functions
   b. "Skip" prohibitive function iff its input is symbolic

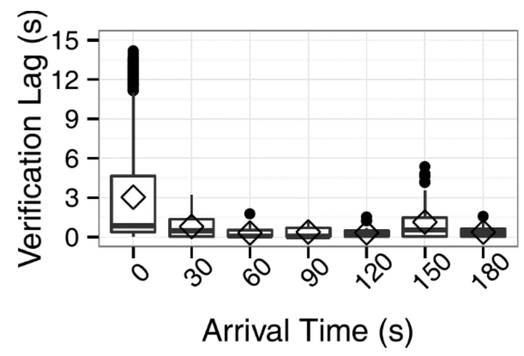
Results

The verifier was evaluated on a system with 3.2GHz Intel Xeon E5-2667v3 cores and 256 GB RAM. Network setup: passive tap.

Real-World Performance Test – Gmail payload on TLS 1.2

Single-CPU verification of TLS layer of a 3-min Gmail session (21 TLS sessions, 3.8MB data)

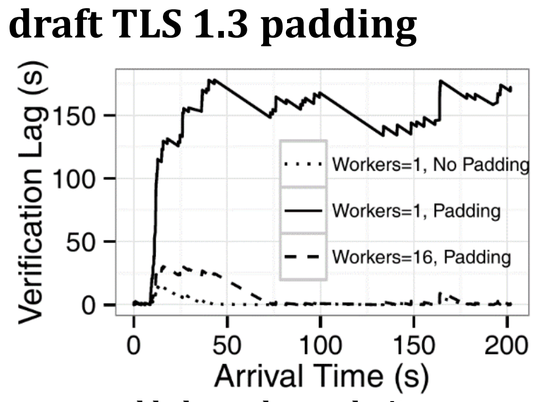
Notation: *lag(n)* = duration between msg_n's arrival and time it is declared valid, given that msg_n verification cannot begin until msg_{n-1} completes. All 21 TLS sessions are summarized. Box plot at horizontal-axis value t includes $\{lag(i): t \le arrival(i) < t + 30s\}$. ◊ = mean.



The verification completes within the wall-clock interval for which the sessions are active.

Stress Test – Added complexity: draft TLS 1.3 padding

TLS 1.3 supports encrypted random padding, increasing the verifier search space. Using multiple threads enables the verifier to overcome the added complexity.



Up to 128 bytes of padding: 16-worker verifier overcomes added search complexity.

Detection of Two Classes of Client Exploits

- CVE-2014-0160: (Heartbleed) Client sends encrypted, invalid length field resulting in sensitive memory disclosure.
- CVE-2015-0205: Client sends correctly formatted messages; but state machine violation enables authentication bypass.

Single-CPU verifier rejects attack traffic in 6.9s and 2.4s, respectively, with no vulnerability-specific configuration. Note: the most relevant metric for verifier speed is accepting legitimate traffic quickly (above)—not rejecting attack traffic quickly.