A System to Verify Network Behavior of Known Cryptographic Clients

Andrew Chi, Robert A. Cochran, Marie Nesfield, Michael K. Reiter, and Cynthia Sturton, *The University of North Carolina at Chapel Hill*

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/chi

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

A System to Verify Network Behavior of Known Cryptographic Clients

Andrew Chi Robert A. Cochran Marie Nesfield Michael K. Reiter Cynthia Sturton
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

Numerous exploits of client-server protocols and applications involve modifying clients to behave in ways that untampered clients would not, such as crafting malicious packets. In this paper, we develop a system for verifying in near real-time that a cryptographic client's message sequence is consistent with its known implementation. Moreover, we accomplish this without knowing all of the client-side inputs driving its behavior. Our toolchain for verifying a client's messages explores multiple candidate execution paths in the client concurrently, an innovation useful for aspects of certain cryptographic protocols such as message padding (which will be permitted in TLS 1.3). In addition, our toolchain includes a novel approach to symbolically executing the client software in multiple passes that defers expensive functions until their inputs can be inferred and concretized. We demonstrate client verification on OpenSSL and BoringSSL to show that, e.g., Heartbleed exploits can be detected without Heartbleed-specific filtering and within seconds of the first malicious packet. On legitimate traffic our verification keeps pace with Gmail-shaped workloads, with a median lag of 0.85s.

1 Introduction

Tampering with clients in client-server protocols or applications is an ingredient in numerous abuses. These abuses can involve exploits on the server directly, or manipulation of application state for which the client is authoritative. An example of the former is the high-profile Heartbleed [14] vulnerability, which enabled a tampered SSL client to extract contents of server memory. An example of the latter is an "invalid command" game cheat that permits a client greater powers in the game [36].

The ideal defense would be to implement formally verified servers that incorporate all necessary input validation and application-specific checking. However, in practice, current production servers have codebases too large to retrofit into a formally verified model (see Sec. 2). Take for example the continued discovery of critical failures of input validation in all major implementations of Transport Layer Security (TLS) [24]. Despite extensive review, it has been difficult to perfectly implement even "simple" input validation [28], let alone all higher-level program logic that could affect authentication or could compromise the integrity and confidentiality of sensitive data [27].

Since it is generally impossible to anticipate all such abuses, in this paper we explore a holistic approach to validating client behavior as consistent with a sanctioned client program's source code. In this approach, a *behavioral verifier* monitors each client message as it is delivered to the server, to determine whether the sequence of messages received from the client so far is consistent with the program the client is believed to be running and the messages that the server has sent to the client (Fig. 1). Performing this verification is challenging primarily because inputs or nondeterministic events at the client may be unknown to the verifier, and thus, the verifier must deduce (via a solver) whether there exist inputs that could have driven the client software to send the messages it did. Furthermore, some of those inputs may be protected by cryptographic guarantees (private keys in asymmetric cryptography), and maliciously crafted fields may themselves be hidden by encryption, as with Heartbleed.

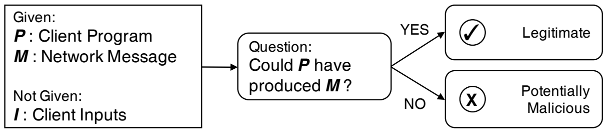


Figure 1: Abstracted behavioral verification problem.

Our central contribution is to show that legitimate cryptographic client behavior can in fact be verified, not against a simplified protocol model but against the client source code. Intuitively, we limit an attacker to only behaviors that could be effected by a legitimate client. We believe this advance to be important: in showing that messages from a client can be quickly verified as legitimate or potentially malicious, we narrow the time between zero-day exploit and detection to mere seconds. This is significant, since in the case of Heartbleed, for example, the bug was introduced in March 2012 and disclosed in April 2014, a window of vulnerability of two years. During this time, few production networks were even monitoring the relevant TLS Heartbeat records, let alone were configured to detect this misbehavior.

Our examination of 70 OpenSSL CVEs from 2014-2016 showed that 23 out of 37 TLS/DTLS server-side vulnerabilities required tampering with feasible client behavior as an ingredient in exploitation. The vulnerabilities comprised input validation failures, memory errors, data leaks, infinite loops, downgrades, and invalid authorization. Our technique accomplishes verification with no vulnerability-specific configuration and, indeed, could have discovered all of these client exploit attempts even prior to the vulnerabilities' disclosure.

Following several other works in verification of client messages when some client-side values are unknown (see Sec. 2), our strategy is to use symbolic execution [7] to trace the client execution based on the messages received so far from the client (and the messages the server has sent to it). When the verifier, in tracing client execution, operates on a value that it does not know (a "symbolic" value), it considers all possibilities for that value (e.g., branching in both directions if a branch statement involves a symbolic variable) and records constraints on those symbolic values implied by the execution path taken. Upon an execution path reaching a message send point in the client software, the verifier reconciles the accumulated constraints on that execution path with the next message received from the client. If the path does not contradict the message, then the message is confirmed as consistent with some valid client execution.

We advance this body of research in two ways.

1. Prior research on this form of client verification has primarily focused on carefully prioritizing candidate paths through the client in the hopes of finding one quickly to validate the message trace observed so far. This prioritization can itself be somewhat expensive (e.g., involving edit-distance computations on execution paths) and prone to error, in which case the verifier's search costs grow dramatically (e.g., [10]). Here we instead use parallelism to explore candidate paths concurrently, in lieu of sophisticated path prediction. In Sec. 6, we highlight one aspect of cryptographic protocols for which efficiently validating client-side behavior depends on being able to explore multiple execution fragments in parallel, namely execution fragments reflecting plaintexts of different sizes, when the true plaintext size is hidden by message padding (as in SSH and draft TLS 1.3). In this case, predicting the plaintext length is not possible from the ciphertext length, by design, and so exploring different candidate lengths in parallel yields substantial savings.
2. When verifying the behavior of a client in a cryptographic protocol such as TLS, the search for a client execution path to explain the next client message can be stymied by paths that contain cryptographic functions for which some inputs are unknown (i.e., symbolic). The symbolic execution of, e.g., the AES block cipher with an unknown message or a modular exponentiation with an unknown exponent is simply too costly. Every message-dependent branch in the modular exponentiation routine would need to be explored, and the large circuit representation of AES would result in unmanageably complex formulas. In Sec. 5 we thus describe a multi-pass algorithm for exploring such paths, whereby user-specified "prohibitive" functions are bypassed temporarily until their inputs can be deduced through reconciliation with the client message; only then is the function explored (concretely). In cases where those inputs can never be inferred—as would be the case for an ephemeral Diffie-Hellman key, for example—the system outputs the assumption required for the verification of the client message to be correct, which can be discharged from a whitelist of assumptions. Aside from these assumptions, our verification is exact: the verifier accepts if and only if the client is compliant.

Our technique, while not completely turnkey, does not require detailed knowledge of the protocol or application being verified. For example, the specification of prohibitive functions and a matching whitelist of permissible assumptions is straightforward in our examples: the prohibitive functions are simply the AES block cipher, hash functions, and elliptic curve group operations; and the whitelist amounts to the assumption that a particular value sent by the client is in the elliptic-curve group (which the server is required to check [26]). Aside from specifying the prohibitive functions and the whitelist, the other needed steps are identifying network send and receive points, minor syntactic modifications to prepare the client software for symbolic execution (see Appendix B), and, optionally, "stubbing out" calls to software that are irrelevant to the analysis (e.g., `printf`). In the case of validating TLS client behavior, we also leverage a common diagnostic feature on servers: logging session keys to enable analysis of network captures.

We show that client verification can coarsely keep pace with a workload equivalent to an interactive Gmail session running over TLS 1.2, as implemented by

OpenSSL and BoringSSL. Verification averages 49ms per TLS record on a 3.2GHz processor, with 85% completing within 36ms and 95% completing within 362ms. Taking into account the bursts of network activity in Gmail traffic, and that a client-to-server record cannot begin verification until all previous client-to-server records are verified, the median verification *lag* between the receipt of a client-to-server record and its successful verification is 0.85s. We also show that our technique similarly keeps pace with TLS 1.2 connections modified to use message padding, a draft TLS 1.3 [30] feature that introduces costs that our parallel approach overcomes.

Our verifier compares client messages against a specific client implementation, and so it is most appropriate in scenarios where an expected client implementation is known. For example, while a plethora of TLS implementations exist on the open internet, only a few TLS clients are likely to be part of a corporate deployment where installed software is tightly controlled. Knowledge of the implementation might also arise by the client revealing its identification string explicitly (e.g., SSH [38]) or by particulars of its handshake (e.g., TLS [1]). That said, we have also made progress on generalizing our verifier to apply across multiple minor revisions (see Sec. 7).

2 Related Work

The most closely related work is due to Bethea et al. [3] and Cochran and Reiter [10]. These works develop algorithms to verify the behavior of (non-cryptographic) client applications in client-server settings, as we do here. Bethea et al. adopted a wholly offline strategy, owing to the expense of their techniques. Cochran and Reiter improved the method by which a verifier searches for a path through the client program that is consistent with the messages seen by the verifier so far. By leveraging a training phase and using observed messages to provide hints as to the client program paths that likely produced those messages, their technique achieved improved verification latencies but still fell far short of being able to keep pace with, e.g., highly interactive games. Their approach would not work for cryptographic protocols such as those we consider here, since without substantial protocol-specific tuning, the cryptographic protections would obscure information in messages on which their technique depends for generating these hints.

Several other works have sought to verify the behavior of clients in client-server protocols. Most permit false rejections or acceptances since they verify client behavior against an abstract (and so imprecise) model of the client program (e.g., [16, 17]), versus an actual client program as we do here. Others seek exact results as we do, but accomplish this by modifying the client to send all inputs it processes to the verifier, allowing the verifier to simply replay the client on those inputs [34]. In our work, we verify actual client implementations and introduce no additional messaging overhead. Proxies for inferring web-form parameter constraints when a web form is served to a client, to detect parameter-tampering attacks when the form values are returned [32], also provide exact detection. However, this work addresses only stateless clients and does so without attention to cryptographically protected traffic. Our work permits stateful clients and specifically innovates to overcome challenges associated with cryptographic protocols.

Also related to our goals are works focused on verifying the correctness of outsourced computations. Recent examples, surveyed by Walfish and Blumberg [35], permit a verifier to confirm (probabilistically) that an untrusted, remote party performed the outsourced computation correctly, at a cost to the verifier that is smaller than it performing the outsourced computation itself. Since we approach the problem from the opposite viewpoint of a well-resourced verifier (e.g., running with the server in a cloud that the server owner trusts), our techniques do not offer this last property. However, ours requires no changes to the party being verified (in our case, the client), whereas these other works increase the computational cost for the party being verified by orders of magnitude (e.g., see [35, Fig. 5]). Another area of focus in this domain has been reducing the privacy impact of the extra information sent to the verifier to enable verification (e.g., [29]). Since our technique does not require changes to the messaging behavior of the application at all, our technique does not suffer from such drawbacks.

More distantly related is progress on proving security of reference implementations of cryptographic protocols relative to cryptographic assumptions (e.g., miTLS, a reference implementation of TLS in F# [5]) or of modules that can be incorporated into existing implementations to ensure subsets of functionality (e.g., state-machine compliance [4]). Our work instead seeks to prove a property of the *messages* in an interaction, namely that these messages are consistent with a specified client implementation. As such, our techniques show nothing about the intrinsic security of the client (or server) implementation itself; nevertheless, they are helpful in detecting a broad range of common exploit types, as we show here. Our techniques are also immediately applicable to existing production protocol implementations.

3 Background and Goals

A client-server protocol generates messages msg_0, msg_1, . . ., some from the client and some sent by the server. Our goal is to construct a *verifier* to validate the client behavior as represented in the message sequence; the server is trusted. We assume that the client is single-threaded and that the message order reflects the order in which the client sent or received those messages, though neither of these assumptions is fundamental. [1] Our technique is not dependent on a particular location for the verifier, though for the purposes of this paper, we assume it is near the server, acting as a passive network tap.

Borrowing terminology from prior work [10], the task of the verifier is to determine whether there exists an *execution prefix* of the client that is *consistent* with the messages $msg_0, msg_1, \ldots$, as defined below.

Definition 1. Execution Prefix. An execution prefix Π is a sequence of client instructions that begins at the client entry point and follows valid branching behavior in the client program. The sequence may include calls to POSIX `send()` and `recv()`, which are considered "network I/O instructions" and denoted SEND and RECV.

Definition 2. Consistency. An execution prefix Π_n is *consistent* with msg_0, msg_1, …, msg_n, iff:

- Π_n contains exactly $n+1$ network I/O instructions $\{\gamma_0, ..., \gamma_n\}$, with possibly other instructions.
- $\forall i \in \{0, \ldots, n\}$, $direction(\gamma_i)$ matches the direction of msg_i, where $direction(\text{SEND})$ is client-to-server and $direction(\text{RECV})$ is server-to-client.
- The branches taken in Π_n were possible under the assumption that msg_0, msg_1,…, msg_n were the messages sent and received.

Consistency of Π_n with msg_0, msg_1,…, msg_n requires that the conjunction of all symbolic postconditions at SEND instructions along Π_n be satisfiable, once concretized using contents of messages msg_0, msg_1,…, msg_n sent and received on that path.

The verifier attempts to validate the sequence msg_0, msg_1, … incrementally, i.e., by verifying the sequence msg_0, msg_1, …, msg_n starting from an execution prefix Π_{n-1} found to be consistent with msg_0, msg_1, …, msg_{n-1}, and appending to it an *execution fragment* that yields an execution prefix Π_n consistent with msg_0, msg_1, …, msg_n. Specifically, an *execution fragment* is a nonempty sequence of client instructions (i) beginning at the client entry point, a SEND, or a RECV in the client software, (ii) ending at a SEND or RECV, and (iii) having no intervening SEND or RECV instructions. If there is no execution fragment that can be appended to Π_{n-1} to produce a Π_n consistent with msg_0, msg_1, …, msg_n, then the search resumes by *backtracking* to find another execution prefix $\hat{\Pi}_{n-1}$ consistent with msg_0, msg_1, …, msg_{n-1}, from which the search resumes for an execution fragment to extend it to yield a $\hat{\Pi}_n$ consistent with msg_0, msg_1, …, msg_n. Only after all such attempts fail can the client behavior be declared invalid.

Determining if a program can output a given value is only semidecidable (recursively enumerable); i.e., while valid client behavior can be declared as such in finite time, invalid behavior cannot, in general. Thus, an "invalid" declaration may require a timeout on the verification process.[2] However, our primary concern in this paper is verifying the behavior of *valid* clients quickly.

4 Parallel Client Verification

As discussed above, upon receipt of message msg_n, the verifier searches for an execution fragment with which to extend execution prefix Π_{n-1} (consistent with msg_0, …, msg_{n-1}) to create an execution prefix Π_n that is consistent with msg_0, …, msg_n. Doing so at a pace that keeps up with highly interactive applications remains a challenge (e.g., [10]). We observe, however, that multiple execution fragments can be explored concurrently. This permits multiple worker threads to symbolically execute execution fragments simultaneously, while coordinating their activities through data structures to ensure that they continue to examine new fragments in priority order. In this section, we give an overview of our parallel verification algorithm; this algorithm is detailed in Appendix A.

In this algorithm, a state σ represents a snapshot of execution in a virtual machine, including all constraints (path conditions) and memory objects, which include the contents (symbolic or concrete) of registers, the stack and the heap. We use $\sigma.\mathsf{cons}$ to represent the constraints accumulated during the execution to reach σ, and $\sigma.\mathsf{nxt}$ to represent the next instruction to be executed from σ. The verifier produces state σ_n by symbolically executing the execution prefix Π_n.

The algorithm builds a binary tree of Node objects. Each node nd has a field nd.path to record a path of instructions in the client; a field nd.state that holds a symbolic state; children fields $\mathsf{nd.child}_0$ and $\mathsf{nd.child}_1$ that point to children nodes; and a field nd.saved that will be described in Sec. 5. The tree is rooted with a node nd holding the state $\mathsf{nd.state} = \sigma_{n-1}$ and $\mathsf{nd.path} = \Pi_{n-1}$. The two children of a node nd in the tree extend nd.path through the next symbolic branch (i.e., branch instruction with a symbolic condition). One child node holds a state with a constraint that maintains that the branch condition implies false, and the other child node's state holds a constraint that indicates that the branch condition

[1]The verifier can optimistically assume the order in which it observes the messages is that in which the client sent or received them, which will often suffice to validate a legitimate client even if not strictly true, particularly when the client-server protocol operates in each direction independently (as in TLS). In other cases, the verifier could explore other orders when verification with the observed order fails. Moreover, several works (e.g., [8, 2]) have made progress on symbolic execution of multi-threaded programs.

[2]Nevertheless, our tool declares our tested exploit traces as invalid within several seconds, after exhaustive exploration of the state space. See Sec. 6.1.

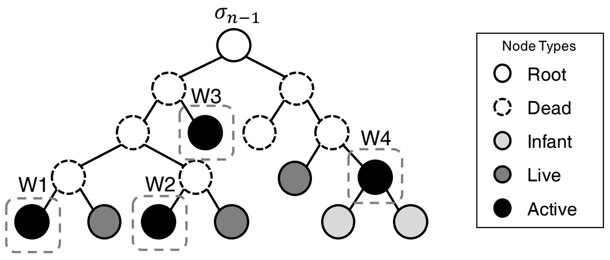


Figure 2: Example node tree.

is true. The algorithm succeeds by finding a fragment with which to extend Π_{n-1} to yield Π_n if, upon extending a path, it encounters a network I/O instruction that yields a state with constraints that do not contradict msg_n being the network I/O instruction's message.

The driving goal of our algorithm is to enable concurrent exploration of multiple states in the node tree. To this end, our algorithm uses multiple threads; one executes a *node scheduler* and the others are *worker* threads, each assigned to one node in the node tree at a time (chosen by the node scheduler, which manages the prioritized heap of unassigned nodes). Fig. 2 shows an example assignment of four workers to multiple nodes in a node tree rooted at σ_{n-1}. White nodes with dashed outlines are *dead* and represent intermediate states that no longer exist. A node is dead if its accumulated constraints reach a contradiction or it has generated children nodes and delivered them to the node scheduler. Black nodes are *active* and are currently being explored by worker threads. Dark-gray nodes are being prioritized by the node scheduler and are still *live*. If there are worker threads that are ready to process a node, they will take their next node from a prioritized queue of the live nodes. Light-gray nodes are *infant* nodes that have just been produced by a worker thread and not yet prioritized by the node scheduler. We can see that worker W4 recently hit a symbolic branch condition and created two infant nodes. The other workers are likely processing straight-line code.

In our design and experiments, the number of worker threads is a fixed parameter provided to the verifier. Because the verification task is largely CPU-bound, in our experience it is not beneficial to use more worker threads than the number of logical CPU cores, and in some cases, fewer worker threads than cores are necessary.

5 Multipass Client Verification

Concurrent exploration of execution fragments can be highly beneficial to the speed of validating legitimate client behavior in cryptographic protocols, as we will show in Sec. 6.3. Nevertheless, there remain challenges to verifying cryptographic clients that no reasonable amount of parallelization can overcome, since doing so would be tantamount to breaking some of the underlying cryptographic primitives. In this section, we introduce a strategy for client verification that can overcome these hurdles for practical protocols such as TLS.

The most obvious challenge is encrypted messages. To make sense of these messages, the verifier needs to be given the symmetric session key under which they are encrypted. Fortunately, existing implementations of, e.g., OpenSSL servers, enable logging session keys to support analysis of network captures, and so we rely on such facilities to provide the session key to the verifier. Given this, it is theoretically straightforward to reverse the encryption on a client-to-server message mid-session—just as the server can—but that capability does surprisingly little to itself aid the verification of the client's behavior, as higher-level protocol logic often composes cryptographic primitives in complex ways. Indeed, state-of-the-art servers routinely fail to detect problems with the message sequence received from a client, as demonstrated by numerous such CVEs [24].

We thus continue with our strategy of incrementally building an execution prefix Π in the client software as each message is received by the verifier to validate the client's behavior. The verifier injects the logged session key into the execution prefix at the point where the key would first be generated by the client. Still, however, the number of execution fragments that need to be explored in cryptographic client implementations is far too large to be overcome by concurrent exploration alone, when other inputs to cryptographic algorithms can be symbolic. Some of these (e.g., a message plaintext, once decrypted) could be injected by the verifier like the session key is, but in our experience, configuring where to inject what values would require much more client-implementation-specific knowledge and bookkeeping than injecting just the session key does. This is in part due to the many layers in which cryptography is applied in modern protocols; e.g., in the TLS handshake, multiple messages are hashed to form the plaintext of another message, which is then encrypted and authenticated. Even worse, other values, e.g., a client's ephemeral Diffie-Hellman key, will never be available to a verifier (or server) and so cannot be injected into an execution prefix.

These observations motivate a design whereby the verifier skips specified functions that would simply be too expensive to execute with symbolic inputs. Specifying such *prohibitive functions* need not require substantial client-implementation-specific or even protocol-specific knowledge; in our experience with TLS, for example, it suffices to specify basic cryptographic primitives such as modular exponentiation, block ciphers, and hash functions as prohibitive. Once specified as prohibitive, the function is skipped by the verifier if any of its inputs are symbolic, producing a symbolic result instead. Once reconciled with the message sequence $msg_0, \ldots, msg_n$ under consideration, however, the verifier can solve for some values that it was previously forced to keep symbolic, after which it can go back and verify function computations (concretely) it had previously skipped. Once additional passes yield no new information, the verifier outputs any unverified function computations (e.g., ones based on the client's ephemeral Diffie-Hellman key) as assumptions on which the verification rests. Only if one of these assumptions is not true will our verifier erroneously accept this message trace. As we will see, these remaining assumptions for a protocol like TLS are minimal.

5.1 User configuration

To be designated prohibitive, a function must meet certain requirements: it must have no side effects other than altering its own parameters (or parameter buffers if passed by reference) and producing a return value; given the same inputs, it must produce the same results; and it must be possible to compute the sizes of all output buffers as a function of the sizes of the input buffers. A function should be specified as prohibitive if it would produce unmanageably complex formulas, such as when the function has a large circuit representation. A function should also be specified as prohibitive if executing it on symbolic inputs induces a large number of symbolic states, due to branching that depends on input values. For example, a physics engine might contain signal processing functions that should be marked prohibitive.

In our case studies, the prohibitive functions are cryptographic functions such as the AES block cipher or SHA-256. We stress, however, that the user need not know how these primitives are composed into a protocol. We illustrate this in Appendix B, where we show the user configuration needed for verifying the OpenSSL client, including the specification of the prohibitive functions. (The configuration for BoringSSL is similar.)

Specifying prohibitive functions generalizes the normal procedure used by symbolic execution to inject symbolic inputs into the program. The user normally designates "user input" functions (such as `getchar`) as symbolic, so that each one is essentially replaced with a function that always returns a symbolic, unconstrained value of the appropriate size. The random number generators, client-side inputs (i.e., `stdin`), and functions that return the current time are also typically so designated. The user configuration for prohibitive functions simply extends this mechanism so that some of these functions do not always return symbolic outputs, but return concrete outputs when their inputs are fully concrete.

5.2 Algorithm overview

The multipass verification algorithm works as follows, when verifying a message msg_n starting from Π_{n-1}. The algorithm expands the binary tree of nodes as described in Sec. 4, with two main differences. First, if the next instruction is a call to a prohibitive function, it is treated as follows: If the prohibitive function is being called with any symbolic input buffers, then its execution is skipped and its outputs are instantiated with fully symbolic output buffers of the appropriate size. If, on the other hand, the prohibitive function is being called with only concrete inputs, then the called function is executed concretely.

Second, upon hitting a network instruction that is consistent with msg_n, the accumulated constraints are saved in a field `nd.saved` for the node `nd` that encountered the network instruction. The execution fragment represented by `nd` is then replayed (starting from Π_{n-1}), again skipping any prohibitive functions encountered with symbolic inputs and concretely executing any encountered with only concrete inputs. Upon hitting the network instruction again, the algorithm compares the previous constraints (saved in `nd.saved`) with the constraints σ.cons accumulated in the re-execution. If no new constraints have been gathered, then additional re-executions of the execution fragment will similarly gather no new constraints. As such, the execution fragment is appended to Π_{n-1} to create Π_n, since it is consistent with all of $msg_0, \ldots, msg_n$, and the algorithm terminates. Any prohibitive functions that were never concretely executed result in an assumption on which the verification rests—specifically, that there is some input to each such prohibitive function that is consistent with the constraints implied by Π_n and $msg_0, \ldots, msg_n$.

Note that the multipass algorithm for msg_n does not reexamine prohibitive functions that were skipped within Π_{n-1}. In cases where this is desired, lazy constraint generation provides a mechanism to do so, as described in Appendix C.

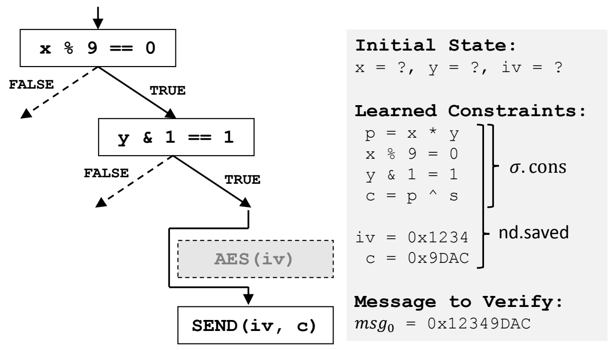
5.3 Detailed walk-through

We now provide a walk-through of this algorithm on a trivial C client shown in Fig. 3a. This client multiplies two of its inputs `x` and `y`, encrypts it using a third input `iv` as an initialization vector, and sends both `iv` and the encrypted value to the server. Our tool begins with a node initialized to the client entry point and attempts to verify (by spawning worker threads) that there exist inputs `x`, `y`, and `iv` that would produce the output message $msg_0 =$ `0x12349DAC` that was observed over the network.

The worker thread that first reaches the SEND has, by that time, accumulated constraints σ.cons as specified in Fig. 3b. Note, however, that it has no constraints relat-

```
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);
    }
  }
}
```

(a) Example client code



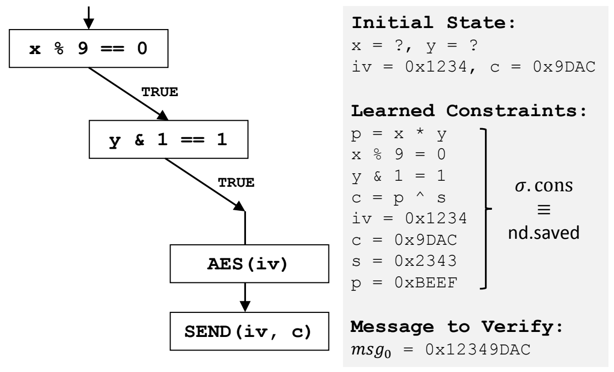
```
Initial State:
x = ?, y = ?, iv = ?

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

iv = 0x1234
 c = 0x9DAC

Message to Verify:
msg_0 = 0x12349DAC
```

(b) Pass one



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

Learned Constraints:
p = x * y
x % 9 = 0
y & 1 = 1
c = p ^ s
iv = 0x1234
c = 0x9DAC
s = 0x2343
p = 0xBEEF

Message to Verify:
msg_0 = 0x12349DAC
```

(c) Pass two

Figure 3: Example of multipass verification on a simple client. σ.cons holds the constraints of the execution path and accepted messages leading to the current state.

ing s (the output of AES(iv)) and iv, since AES was designated as prohibitive and skipped (since iv is symbolic). After reconciling these constraints with the message $msg_0 =$ 0x12349DAC, the verifier records nd.saved.

The verifier then re-executes from the root (Fig. 3c). Since it now knows iv = 0x1234, this time it does not skip AES and so computes s = 0x2343 and 0x9DAC = p ^ 0x2343, i.e., p = 0xBEEF. After this pass, the constraints in nd.saved are still satisfiable (e.g., x = 0x9, y = 0x1537). A third pass would add no new information, and so the thread updates the corresponding execution prefix (nd.path) and state (nd.state).

5.4 TLS example

We illustrate the behavior of the multipass algorithm on TLS. Fig. 4 shows an abstracted subset of a TLS client implementation of AES-GCM, running on a single block of plaintext input. For clarity, the example omits details such as the implicit nonce, the server ECDH parameters, the generation of the four symmetric keys, and subsumes the tag computation into the GHASH function. But in all features shown, this walkthrough closely exemplifies the multi-pass verification of a real-world TLS client.

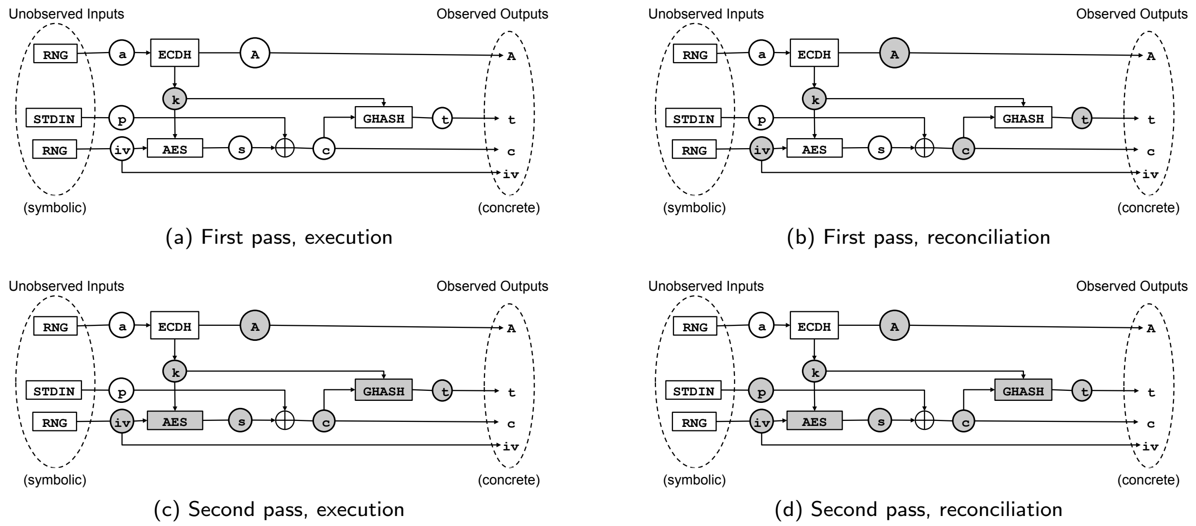
In Fig. 4, the outputs observed by the verifier are the client Diffie-Hellman parameter A, the initialization vector iv, the ciphertext c, and the AES-GCM tag t. The unobserved inputs are the Diffie-Hellman private exponent a, the initialization vector iv, and the plaintext p. We do assume access to the AES symmetric key k. Since client verification is being performed on the server end of the connection, we can use server state, including k.

In the first pass of symbolic execution (Fig. 4a), even with knowledge of the AES symmetric key k, all prohibitive functions (ECDH, AES, GHASH) have at least one symbolic input. So, the verifier skips them and produces unconstrained symbolic output for each. After the first execution pass (Fig. 4b), the verifier encounters the observed client outputs. Reconciling them with the accumulated constraints σ.cons yields concrete values for A, t, c, and iv, but not the other variables.

The verifier then begins the second pass of symbolic execution (Fig. 4c). Now, AES and GHASH both have concrete inputs and so can be executed concretely. The concrete execution of AES yields a concrete value for s, which was not previously known. After the second execution pass (Fig. 4d), the verifier implicitly uses the new knowledge of s to check that there is a p, the unobserved plaintext value, that satisfies the constraints imposed by observed output. Further passes yield no additional information, as no further symbolic inputs to prohibitive functions can be concretized.

Note that the value of a, the client Diffie-Hellman private exponent, is never computed. The verifier thus outputs an assumption that there exists an a such that ECDH(a) yields values A and k. As such, we do not detect invalid curve attacks [19], for example; we discuss practical mitigations for this in Sec. 7.3. See Appendix B for the whitelisting of this assumption for a real TLS client.

Note that no decryption mechanism is provided to the verifier. The multipass mechanism automatically recovers the plaintext for stream ciphers and counter-mode block ciphers such as AES-GCM. For other, less preferred modes such as CBC, the user can provide inverse functions via a feature described in Appendix C.



(a) First pass, execution (b) First pass, reconciliation (c) Second pass, execution (d) Second pass, reconciliation

Figure 4: 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. In Fig. 4b and Fig. 4d, some values become concrete when σ.cons is reconciled with msg_n.

6 Evaluation

In this section we evaluate our implementation of the algorithms in Secs. 4–5. Our implementation is built on a modified version of KLEE [9], a symbolic execution engine for LLVM assembly instructions, and leverages KLEE's POSIX model, an expanded POSIX model used by Cloud9 [8], and our own model of POSIX network calls. We applied our implementation to verify OpenSSL and BoringSSL clients. BoringSSL, a fork of OpenSSL aiming to improve security and reduce complexity, incorporates changes to the API, thread safety, error handling, and the protocol state machine—resulting in a code base of 200,000 lines of code vs. OpenSSL's 468,000. BoringSSL has been independently maintained since June 2014, and is now deployed throughout Google's production services, Android, and Chrome [23].

Our evaluation goals were as follows. First, we ran a one-worker verifier against two attacks on OpenSSL that represent different classes of client misbehavior, to illustrate detection speed. Second, we load tested a single-worker verifier on a typical TLS 1.2 payload—the traffic generated by a Gmail session—to illustrate the performance of verifying legitimate client behavior. Third, we increased the verification complexity to demonstrate scalability to more complex protocols with larger client state spaces, which we overcome using multiple workers. We did this by verifying a TLS 1.3 draft [30] feature that permits random padding in every packet. The OpenSSL instrumentation (203 lines) and verifier configuration options (138 lines) we used are described in Appendix B; the BoringSSL setup was similar.

The experiments were run on a system with 3.2GHz Intel Xeon E5-2667v3 processors, with peak memory usage of 2.2GB. For the majority of the experiments, a single core ran at 100% utilization. The only exception to this was the third set of experiments (random padding), where up to 16 cores were allocated, though actual utilization varied significantly depending on workload.

Our main performance measure was verification *lag*. To define lag, let the *cost* of msg_n, denoted $cost(n)$, be the wall-clock time that the verifier spends to conclude if msg_n is valid, beginning from an execution prefix Π_{n-1} consistent with $msg_0, \ldots, msg_{n-1}$. That is, $cost(n)$ is the time it spends to produce Π_n from Π_{n-1}. The *completion time* for msg_n is then defined inductively as follows:

$$comp(0) = cost(0)$$
$$comp(n) = \max\{arr(n), comp(n-1)\} + cost(n)$$

where $arr(n)$ is the wall-clock time when msg_n arrived at the verifier. Since the verification of msg_n cannot begin until after both (i) it is received at the verifier (at time $arr(n)$) and (ii) the previous messages $msg_0, \ldots, msg_{n-1}$ have completed verification (at time $comp(n-1)$), $comp(n)$ is calculated as the cost $cost(n)$ incurred after both (i) and (ii) are met. Finally, the *lag* of msg_n is $lag(n) = comp(n) - arr(n)$.

6.1 Misbehavior detection

We first evaluated our client verifier against two attacks on OpenSSL that are illustrative of different classes of vulnerabilities that we can detect: those related to tampering with the client software to produce messages that a client could not have produced (CVE-2014-0160 Heartbleed) and those with message sequences that, while correctly formatted, would be impossible given a valid client state machine (CVE-2015-0205). Note that testing client *misbehavior* required proof-of-concept attacks, usually prudently omitted from CVEs. We therefore constructed our own attack against each vulnerability and confirmed that each attack successfully exploited an appropriately configured server.

An OpenSSL `1.0.1f` `s_server` was instantiated with standard settings, and an OpenSSL `s_client` was modified to establish a TLS connection and send a single Heartbleed exploit packet. This packet had a modified length field, and when received by an OpenSSL `1.0.1f` `s_server`, caused the server to disclose sensitive information from memory. When the trace containing the Heartbleed packet was verified against the original OpenSSL `1.0.1f` `s_client`, the verifier rejected the packet after exhausting all search paths, with a lag for the Heartbleed packet of 6.9s.

Unlike Heartbleed, CVE-2015-0205 involved only correctly formatted messages. In the certificate exchange, a good client would send a DH certificate (used to generate a pre-master secret), followed by an empty ClientKeyExchange message. A malicious client might send a certificate followed by a ClientKeyExchange message containing a DH parameter. The server would then authenticate the certificate but prefer the second message's DH parameter, allowing a malicious client to impersonate anyone whose public certificate it obtained.

The verifier rejected an attempted attack after a lag of 2.4s, exhausting the search space. This exploit illustrates the power of our technique: we not only verify whether each message is possible in isolation, but also in the context of all previous messages.

Since the tool verifies *valid* client behavior, no attack-specific configuration was required. We do not require any foreknowledge of the exploit and anticipate correct detection of other exploits requiring client tampering.

6.2 Performance evaluation

Our Gmail performance tests measured the lag that resulted from running single-worker verifiers against real-world TLS traffic volumes. The data set was a `tcpdump` capture of a three-minute Gmail session using Firefox, and consisted of 21 concurrent, independent TLS sessions, totaling 3.8MB of network data. This Gmail session was performed in the context of one of the authors' email accounts and included both receiving emails and sending emails with attachments.

In this test we verified the TLS layer of a network connection, but not the application layer above it, such as the browser logic and Gmail web application. To simulate the client-server configuration without access to Gmail servers and private keys, we used the packet sizes and timings from the Gmail `tcpdump` to generate 21 equivalent sessions using OpenSSL `s_client` and `s_server` and the BoringSSL equivalents, such that the amount of traffic sent in each direction at any point in time matched identically with that of the original Gmail capture.[3] The plaintext payload (Gmail web application data) of each session was also replayed exactly, though the payload contents were unlikely to affect TLS performance. One of the 21 TLS sessions was responsible for the vast majority of the data transferred, and almost all of the data it carried was from the server to the client; presumably this was a bulk-transfer connection that was involved in prefetching, attachment uploading, or other latency-insensitive tasks. The other 20 TLS sessions were utilized more lightly and presumably involved more latency-sensitive activities. Since `s_client` implements a few diagnostic features in addition to TLS (but no application layer), verifying `s_client` against these 21 sessions provided a conservative evaluation of the time required to verify the pure TLS layer.

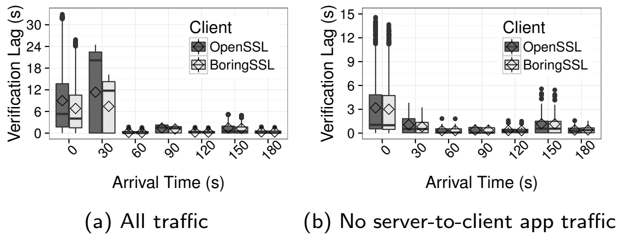


Figure 5: Verification lags for Gmail data set. Box plot at arrival time t includes $\{lag(i) : t \leq arr(i) < t + 30\text{s}\}$. Fig. 5a shows lags for all messages in all 21 TLS sessions. Fig. 5b shows lags if server-to-client application-data messages are dropped.

Fig. 5 shows the distribution of verification lag of messages, grouped by the 30-second interval in which they arrived at the verifier. In each box-and-whisker plot, the three horizontal lines making up each box represent the first, second (median), and third quartiles, and the whiskers extend to cover points within 1.5× the interquartile range. Outliers are shown as single points. In addition, the diamond shows the average value. Fig. 5a show all of the messages' verification lag. It is evident from these figures that the majority of the verification lag

[3]To confirm the appropriateness of using the BoringSSL `s_client`-equivalent in these experiments, we also used it in verification of an unmodified Chrome v50.0.2661.75 browser interacting with an Apache HTTP server.

happened early on, initially up to $\sim$ 30s in the worst case. This lag coincided with an initial burst of traffic related to requests while loading the Gmail application. Another burst occurred later, roughly 160s into the trace, when an attachment was uploaded by the client. Still, the lag for all sessions was near zero in the middle of the session and by the end of the session, meaning that verification for all sessions (in parallel) completed within approximately the wall-clock interval for which the sessions were active.

Verification cost averaged 49ms per TLS record, with 85% costing $\leq$36ms and 95% costing $\leq$362ms. Fig. 6 details cost per message size for all 21 TLS sessions. Despite being smaller, client-to-server messages are costlier to verify, since the verifier's execution of the client software when processing server-to-client messages is almost entirely concrete. In contrast, the execution of the client in preparation of sending a client-to-server message tends to involve more symbolic branching. Also, note the linearity of the relationship between message size and verification cost, particularly for client-to-server messages. This feature suggests a simple, application-independent way to estimate the verification costs for TLS sessions carrying payloads other than Gmail. Assuming similar message sizes in each direction, a deployment could set a sharp timeout at which point the verifier declares the client "invalid." For example, if Fig. 6 were a representative sample of the workloads in a deployment, it would indicate that setting a timeout at a mere 2s (verification cost) could allow the verifier to quickly detect misbehaving clients at a vanishingly small false alarm rate.

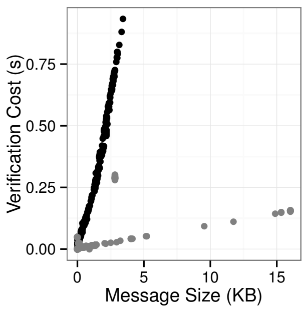


Figure 6: Size versus cost for client-to-server (●) and server-to-client (●) messages.

TLS-Specific Optimizations. While our goal so far had been to provide for client behavior verification with a minimum of protocol-specific tuning, a practical deployment should leverage properties of the protocol for performance. One important property of TLS (and other TCP-based protocols such as SSH) is that its client-to-server and server-to-client message streams operate independently. That is, with the exception of the initial session handshake and ending session teardown, the verifiability of client-to-server messages should be unaffected by which, if any, server-to-client messages the client has received. This gives the verifier the freedom to simply ignore server-to-client application data messages. By doing so, the verification costs for server-to-client messages, now effectively reduced to zero, did not contribute to a growing lag. The effect of this optimization on lag is shown in Fig. 5b, in particular reducing the median lag to 0.85s and the worst-case lag to around 14s. In all subsequent results, we have ignored server-to-client messages unless otherwise noted.

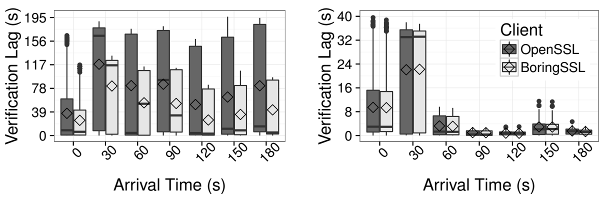
6.3 Stress testing: Added complexity

The Gmail performance evaluation showed that verification of a typical TLS 1.2 session can be done efficiently and reliably, an advance made possible by applying a multipass methodology to cryptographic functions. In essence, once the state explosion from cryptographic functions is mitigated, the client state space becomes small enough that the verification time is primarily determined by the straight-line execution speed of the KLEE symbolic interpreter. However, not all clients are guaranteed to be this simple. One good example is the draft TLS 1.3 standard [30]. In order to hide the length of the plaintext from an observer, implementations of TLS 1.3 are permitted (but not required) to pad an encrypted TLS record by an arbitrary size, up to maximum TLS record size. This random encrypted padding hides the size of the plaintext from any observer, whether an attacker or a verifier. In other words, given a TLS 1.3 record, the length of the input (e.g., from `stdin`) that was used to generate the TLS record could range anywhere from 0 to the record length minus header. Other less extreme examples of padding include CBC mode ciphers, and the SSH protocol, in which a small amount of padding protects the length of the password as well as channel traffic.

We extended our evaluation to stress test our verifier beyond typical current practice. We simulated the TLS 1.3 padding feature by modifying a TLS 1.2 client (henceforth designated "TLS 1.2+"), so that each TLS record includes a random amount of padding up to 128 bytes[4], added before encryption. We then measured verification performance, ignoring server-to-client messages (except during session setup and teardown) as before.

Fig. 7 shows the performance of our single- and 16-worker verifiers on TLS 1.2+ with a random amount of encrypted padding. The addition of random padding to TLS 1.2+ significantly enlarges the client state space that must be explored. With a single-worker verifier, the verification cost increases substantially compared to the TLS 1.2 baseline. The 16-worker verifier reduces the verification cost nearly back to the TLS 1.2 baseline levels. This demonstrates that the state space search is highly amenable to parallelization.

[4]While 128 bytes of padding may seem extreme, previous work showed that an attacker could sometimes infer the website visited by encrypted HTTP connections even with substantial padding (e.g., [25]).



(a) NumWorkers = 1 (b) NumWorkers = 16

Figure 7: Verification lags for Gmail data set when up to 128 bytes of padding are added to each application plaintext, over all 21 TLS sessions. Box plot at arrival time t includes $\{lag(i) : t \leq arr(i) < t + 30s\}$. Fig. 7a shows the lag for a one-worker verifier, and Fig. 7b shows the lag for a 16-worker verifier.

7 Discussion

Here we discuss an approach for dealing with multiple client versions, the applications for which our design is appropriate, and several limitations of our approach.

7.1 Multi-version verifiers

When the version of the client software used by the verifier differs from the version being run by a legitimate client, it is possible for the verifier to falsely accuse the client of being invalid. This poses a challenge for verification when the client version is not immediately evident to the verifier. For example, TLS does not communicate the minor version number of its client code to the server. The possibility for false accusations here is real: we confirmed, e.g., that a verifier for OpenSSL client `1.0.1e` can fail if used to verify traffic for OpenSSL client `1.0.1f`, and vice versa. This occurs because, e.g., the changes from `1.0.1e` to `1.0.1f` included removing MD5 from use and removing a timestamp from a client nonce, among other changes and bug fixes. In total, `1.0.1f` involved changes to 102 files amounting to 1564 insertions and 997 deletions (according to `git`), implemented between Feb 11, 2013 and Jan 6, 2014.

One solution to this problem is to run a verifier for any version that a legitimate client might be using. By running these verifiers in parallel, a message trace can be considered valid as long as one verifier remains accepting of it. Running many verifiers in parallel incurs considerable expense, however.

Another approach is to create one verifier that verifies traffic against several versions simultaneously—a *multi-version verifier*—while amortizing verification costs for their common code paths across all versions. To show the potential savings, we built a multi-version verifier for both `1.0.1e` and `1.0.1f` by manually assembling a "unioned client" of these versions, say "`1.0.1ef`". In client `1.0.1ef`, every difference in the code between client `1.0.1e` and client `1.0.1f` is preceded by a branch on version number, i.e.,

```
if (strcmp(version, "1.0.1e") == 0) {
   /* 1.0.1e code here */
} else {
   /* 1.0.1f code here */
}
```

We then provided this as the client code to the verifier, marking `version` as symbolic. Note that once the client messages reveal behavior that is consistent with only one of `1.0.1e` and `1.0.1f`, then `version` will become concrete, causing the verifier to explore only the code paths for that version; as such, the verifier still allows only "pure `1.0.1e`" or "pure `1.0.1f`" behavior, not a combination thereof.

The single-worker costs (specifically, $\sum_i cost(i)$) of verifying `1.0.1e` traffic with a `1.0.1ef` verifier and of verifying `1.0.1f` traffic with a `1.0.1ef` verifier were both within 4% of the costs for verifying with a `1.0.1e` and `1.0.1f` verifier, respectively. (For these tests, we used the same Gmail traces used in Sec. 6.) Despite a 32% increase in symbolic branches and a 7% increase in SMT solver queries, the overall cost increases very little. This implies that despite an increase in the number of code path "options" comprising the union of two versions of client code, the incorrect paths die off quickly and contribute relatively little to total verification cost, which is dominated by straight-line symbolic execution of paths common to both versions.

While a demonstration of a multi-version verifier for only two versions of one codebase, we believe this result suggests a path forward for verifying clients of unknown versions much more efficiently than simply running a separate verifier for each possibility. We also anticipate that multi-version verifiers can be built automatically from commit logs to repositories, a possibility that we hope to explore in future work.

7.2 Applications

Suitable Application Layers. Consider a deployment of behavioral verification as an intrusion detection system (IDS). Verification lag determines the period that a server does not know a message's validity. The application layer chosen for our TLS evaluation, Gmail, exhibited relatively high lag due to its high-volume data transfers. Other applications may be more optimal for behavioral verification. For example, XMPP [31] generally sends small XML payloads for text-based Internet messaging. Another setting is electronic mail (SMTP) [21], which originally lacked security. Gradually [15], the internet community has deployed mechanisms such STARTTLS [18], SPF [20], DKIM [12], and DMARC [22], many with cryptographic guarantees of authenticity. Behavioral verification can provide a strong compliance check of these mechanisms. Although data volumes can be large, the application is tolerant of delay, making verification lag acceptable.

Other Cryptographic Protocols. Perhaps due to its use in various applications, TLS is one of the more complex security protocols. We believe that the client verification technique should generalize to other, often simpler, protocols. One example is Secure Shell (SSH) [37, 38]. When used as a remote shell, SSH requires low latency but transfers a relatively small amount of data: key presses and terminal updates. When used for file transfer (SFTP), a large volume of data is sent, but in a mode that is relatively latency-insensitive.

7.3 Limitations

Source Code and Configuration. Our verifier requires the client source code to generate LLVM bitcode and to designate prohibitive functions. We also require knowledge of the client configuration, such as command line parameters controlling the menu of possible cipher suites. Again, our approach is most suitable for environments with a known client and configuration.

Environment Modeling. While OpenSSL `s_client` has relatively few interactions with the environment, other clients may interact with the environment extensively. For example, SSH reads `/etc/passwd`, the `.ssh/` directory, redirects standard file descriptors, etc. The KLEE [9] and Cloud9 [8] POSIX runtimes serve as good starting points, but some environment modeling is likely to be necessary for each new type of client. This one-time procedure per client is probably unavoidable.

Manual Choice of Prohibitive Functions. We currently choose prohibitive functions manually. The choice of hash functions, public key algorithms, and symmetric ciphers may be relatively obvious to security researchers, but not necessarily to a typical software developer.

Prohibitive Function Assumptions. When prohibitive functions are initially skipped but eventually executed concretely, verification soundness is preserved. If a prohibitive function is never executed concretely (e.g., due to asymmetric cryptography), this introduces an assumption; e.g., in the case of ECDH, a violation of this assumption could yield an invalid curve attack [19]. In a practical deployment, the user designating a prohibitive function should also designate predicates on the function's output (e.g., the public key is actually a group element) that are specified by the relevant NIST or IETF standards as mandatory server-side checks [26] (which would have prevented the Jager et al. attack [19]). In our tool, these predicates could be implemented via lazy constraint generation (see Appendix C), or as a `klee_assume` for simple predicates. We recommend typical precautions [13] to avoid Bleichenbacher-type attacks [6].

Denial of Service. We anticipate our verifier being deployed as an IDS via a passive network tap. To mitigate a potential denial of service (DoS) attack, one could leverage the linear relationship between verification cost and message size: (1) Impose a hard upper bound on verifier time per packet, and declare all packets that exceed the time budget invalid. Since our results show legitimate packets finish within a few seconds, the bound could easily be set such that the false alarm rate is negligible. (2) Given a fixed CPU time budget, precisely compute the amount of traffic that can be verified. The operator could then allocate verifiers according to the threat profile, e.g., assigning verifiers to high-priority TLS sessions or ones from networks with poor reputation (e.g., [11]). This would degrade verification gracefully as total traffic bandwidth grows beyond the verification budget.

8 Conclusion

We showed that it is possible to practically verify that the messaging behavior of an untrusted cryptographic client is consistent with its known implementation. Our technical contributions are twofold. First, we built a parallel verification engine that supports concurrent exploration of paths in the client software to explain a sequence of observed messages. This innovation is both generally useful for client verification and specifically useful for verifying cryptographic clients, e.g., due to ambiguities arising from message padding hidden by encryption. Second, we developed a multipass verification strategy that enables verification of clients whose code contains cryptographic functions, which typically pose major challenges to symbolic execution. We demonstrated that our verifier detects two classes of client misbehavior: those that produce malformed messages, and those whose message sequence is impossible. In addition, we showed that our verifier can coarsely keep pace with a Gmail TLS workload, running over both OpenSSL and BoringSSL TLS 1.2 and over a more complex simulation of TLS 1.3. We believe our system could dramatically reduce the detection time of protocol exploits, with no prior knowledge of the vulnerabilities.

Acknowledgements. We thank our shepherd, Boon Thau Loo, and the anonymous reviewers for their comments. This work was supported in part by NSF grants 1115948 and 1330599, grant N00014-13-1-0048 from the Office of Naval Research, and a gift from Cisco.

References

[1] B. Anderson, S. Paul, and D. A. McGrew. Deciphering malware's use of TLS (without decryption). *arXiv preprint*, abs/1607.01639, 2016.

[2] T. Bergan, D. Grossman, and L. Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. In *2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 491–506, 2014.

[3] D. Bethea, R. A. Cochran, and M. K. Reiter. Server-side verification of client behavior in online games. *ACM Transactions on Information and System Security*, 14(4), Dec. 2011.

[4] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, P.-Y. S. A. Pironti, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *36th IEEE Symposium on Security and Privacy*, pages 535–552, 2015.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub. Implementing TLS with verified cryptographic security. In *34th IEEE Symposium on Security and Privacy*, pages 445–459, 2013.

[6] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*. 1998.

[7] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *International Conference on Reliable Software*, pages 234–245, 1975.

[8] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *6th European Conference on Computer Systems*, 2011.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[10] R. A. Cochran and M. K. Reiter. Toward online verification of client behavior in distributed applications. In *20th ISOC Network and Distributed System Security Symposium*, 2013.

[11] M. P. Collins, T. J. Shimeall, S. Faber, J. Janies, R. Weaver, M. De Shon, and J. Kadane. Using uncleanliness to predict future botnet addresses. In *7th Internet Measurement Conference*, pages 93–104, 2007.

[12] D. Crocker, T. Hansen, and M. Kucherawy. DomainKeys Identified Mail (DKIM) Signatures. RFC 6376 (Internet Standard), Sept. 2011.

[13] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176, 7465, 7507.

[14] K. Durumeric, J. Kasten, D. Adrian, A. J. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of Heartbleed. In *Internet Measurement Conference*, 2014.

[15] Z. Durumeric, D. Adrian, A. Mirian, J. Kasten, E. Bursztein, N. Lidzborski, K. Thomas, V. Eranti, M. Bailey, and J. A. Halderman. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *2015 ACM Internet Measurement Conference*, pages 27–39, 2015.

[16] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, Aug. 2002.

[17] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *18th International World Wide Web Conference*, pages 561–570, Apr. 2009.

[18] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207 (Proposed Standard), Feb. 2002. Updated by RFC 7817.

[19] T. Jager, J. Schwenk, and J. Somorovsky. Practical invalid curve attacks on TLS-ECDH. In *Computer Security – ESORICS 2015*, volume 9326 of *Lecture Notes in Computer Science*. 2015.

[20] S. Kitterman. Sender Policy Framework (SPF) for authorizing use of domains in email, version 1. RFC 7208 (Proposed Standard), Apr. 2014. Updated by RFC 7372.

[21] J. Klensin. Simple Mail Transfer Protocol. RFC 5321 (Draft Standard), Oct. 2008. Updated by RFC 7504.

[22] M. Kucherawy and E. Zwicky. Domain-based Message Authentication, Reporting, and Conformance (DMARC). RFC 7489 (Informational), Mar. 2015.

[23] A. Langley. BoringSSL. ImperialViolet, Oct. 2015. `https://www.imperialviolet.org/2015/10/17/boringssl.html`.

[24] J. Leyden. Annus HORRIBILIS for TLS! all the bigguns now officially pwned in 2014. The Register, Nov. 2014. `http://www.theregister.co.uk/2014/11/12/ms_crypto_library_megaflaw/`.

[25] M. Liberatore and B. N. Levine. Inferring the source of encrypted HTTP connections. In *13th ACM Conference on Computer and Communications Security*, pages 255–263, 2006.

[26] D. McGrew, K. Igoe, and M. Salter. Fundamental elliptic curve cryptography algorithms. RFC 6090 (Proposed Standard), Feb. 2011.

[27] MITRE. Divide-and-conquer session key recovery in SSLv2 (OpenSSL). CVE-2016-0703, Mar. 1 2016. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0703`.

[28] MITRE. Memory corruption in the ASN.1 encoder (OpenSSL). CVE-2016-2108, May 3 2016. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2108`.

[29] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, May 2013.

[30] E. Rescorla. The Transport Layer Security (TLS) protocol version 1.3. Internet-Draft draft-ietf-tls-tls13-18 (work in progress), IETF Secretariat, October 2016.

[31] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): core. RFC 6120 (Proposed Standard), Mar. 2011.

[32] N. Skrupsky, P. Bisht, T. Hinrichs, V. N. Venkatakrishnan, and L. Zuck. TamperProof: A server-agnostic defense for parameter-tampering attacks on web applicatoins. In *3rd ACM Conference on Data and Application Security and Privacy*, Feb. 2013.

[33] S. Vaudenay. Security flaws induced by CBC padding - Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology - EUROCRYPT 2002*, pages 534–546, 2002.

[34] K. Vikram, A. Prateek, and B. Livshits. Ripley: Automatically securing Web 2.0 applications through replicated execution. In *16th ACM Conference on Computer and Communications Security*, Nov. 2009.

[35] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), Feb. 2015.

[36] S. Webb and S. Soh. A survey on network game cheats and P2P solutions. *Australian Journal of Intelligent Information Processing Systems*, 9(4):34–43, 2008.

[37] T. Ylonen and C. Lonvick. The Secure Shell (SSH) authentication protocol. RFC 4252 (Proposed Standard), Jan. 2006.

[38] T. Ylonen and C. Lonvick. The Secure Shell (SSH) transport layer protocol. RFC 4253 (Proposed Standard), Jan. 2006. Updated by RFC 6668.

A Algorithm Details

The algorithm for verifying a client-to-server message works as follows. This algorithm, denoted $\mathsf{ParallelVerify}$, takes as input the execution prefix Π_{n-1} consistent with $msg_0, \ldots, msg_{n-1}$; the symbolic state σ_{n-1} resulting from execution of Π_{n-1} from the client entry point on message trace $msg_0, \ldots, msg_{n-1}$; and the next message msg_n. Its output is Rslt, which holds the prefix Π_n and corresponding state σ_n in $\mathsf{Rslt.path}$ and $\mathsf{Rslt.state}$, respectively, if a prefix consistent with $msg_0, \ldots, msg_n$ is found. If the procedure returns with $\mathsf{Rslt.path} = \mathsf{Rslt.state} = \perp$, then this indicates that there is no execution prefix that can extend Π_{n-1} to make Π_n that is consistent with msg_0, …, msg_n. This will induce backtracking to search for another $\hat{\Pi}_{n-1}$ that is consistent with msg_0, …, msg_{n-1}, which the verifier will then try to extend to find a $\hat{\Pi}_n$ consistent with msg_0, …, msg_n.

A.1 Parallel verification

$\mathsf{ParallelVerify}$ runs in a thread that spawns $\mathsf{NumWorkers} + 1$ child threads: one thread to manage scheduling of nodes for execution via the procedure $\mathsf{NodeScheduler}$ (not shown) and $\mathsf{NumWorkers}$ worker threads to explore candidate execution fragments via the procedure VfyMsg (Fig. 8).

$\mathsf{NodeScheduler}$ manages the selection of node states to execute next and maintains the flow of nodes between worker threads. It receives as input two queues of nodes, a "ready" queue Q_R and an "added" queue Q_A. These queues are shared between the worker threads and the $\mathsf{NodeScheduler}$ thread. Worker threads pull nodes from Q_R and push new nodes onto Q_A. As there is only one scheduler thread and one or more worker threads producing and consuming nodes from the queues Q_R and Q_A, Q_R is a single-producer-multi-consumer priority queue and Q_A is a multi-producer-single-consumer queue. The goal of NodeScheduler is to keep Q_A empty and Q_R full. Nodes are in one of four possible states, either actively being explored inside VfyMsg, stored in Q_R, stored in Q_A, or being prioritized by NodeScheduler. A node at the front of Q_R is the highest priority node not currently being explored. The nodes in Q_A are child nodes that have been created by VfyMsg threads that need to be prioritized by NodeScheduler and inserted into Q_R. NodeScheduler continues executing until the boolean Done is set to true by some VfyMsg thread.

Shown in Fig. 8, the procedure VfyMsg does the main work of client verification: stepping execution forward in the state σ of each node. In this figure, lines shaded gray will be explained in Sec. A.2 and can be ignored for now (i.e., read Fig. 8 as if these lines simply do not exist). VfyMsg runs inside of a while loop until the value of Done is no longer equal to **false** (101). Recall that the parent procedure ParallelVerify spawns multiple instances of VfyMsg. Whenever there is a node on the queue Q_R, the condition on line 102 will be true and the procedure calls dequeue atomically. Note that even if $|\mathsf{Q}_R| = 1$, multiple instances of VfyMsg may call dequeue in 103, but only one will return a node; the rest will retrieve undefined ($\perp$) from dequeue.

If nd is not undefined (104), the algorithm executes the state nd.state and extends the associated path nd.path up to either the next network instruction (SEND or RECV) or the next symbolic branch (a branch instruction that is conditioned on a symbolic variable). The first case, stepping execution on a non-network / non-symbolic-branch instruction σ.nxt (here denoted isNormal(σ.nxt)), executes in a while loop on lines 106–108. The current instruction is appended to the path and the procedure execStep is called, which symbolically executes the next instruction in state σ. These lines are where most of the computation work is done by the verifier. Concurrently stepping execution on multiple states is where the largest performance benefits of parallelization are achieved. Note that calls to execStep may invoke branch instructions, but these are non-symbolic branches.

In the second case, if the next instruction is SEND or RECV and if the constraints σ.cons accumulated so far with the symbolic state σ do not contradict the possibility that the network I/O message σ.nxt.msg in the next instruction σ.nxt is msg_n (i.e., $(\sigma.\mathsf{cons} \wedge \sigma.\mathsf{nxt.msg} = msg_n) \not\Rightarrow$ **false**, line 110), then the algorithm has successfully reached an execution prefix Π_n consistent with msg_0, …, msg_n. The algorithm sets the termination value (Done = **true**) and sets the return values of the parent function on lines 112–113: Rslt.path is set to

```
100 procedure VfyMsg($msg_n$, Root, $Q_R$, $Q_A$, Done, Rslt)
101   while ¬Done do
102     if $|Q_R| > 0$ then
103       nd ← dequeue($Q_R$)
104       if nd ≠ ⊥ then
105         $\pi$ ← nd.path ; $\sigma$ ← nd.state
106         while isNormal($\sigma$.nxt) do
107           $\pi \leftarrow \pi \,\|\, \langle\sigma.\mathsf{nxt}\rangle$
108           $\sigma$ ← execStep($\sigma$)
109         if isNetInstr($\sigma$.nxt) then
110           if $(\sigma.\mathsf{cons} \wedge \sigma.\mathsf{nxt.msg} = msg_n) \not\Rightarrow$ false then
111             if $(\sigma.\mathsf{cons} \wedge \sigma.\mathsf{nxt.msg} = msg_n) \equiv$ nd.saved then
112               Rslt.path ← $\pi \,\|\, \langle\sigma.\mathsf{nxt}\rangle$
113               Rslt.state ← [execStep($\sigma$) | $\sigma$.nxt.msg ↦ $msg_n$]
114               Done ← true                                    ▷ Success!
115             else
116               nd ← clone(Root)
117               nd.saved ← $\sigma.\mathsf{cons} \wedge \sigma.\mathsf{nxt.msg} = msg_n$
118               enqueue($Q_A$, nd)
119         else if isProhibitive($\sigma$.nxt) then
120           nd.path ← $\pi \,\|\, \langle\sigma.\mathsf{nxt}\rangle$
121           nd.state ← execStepProhibitive($\sigma$, nd.saved)
122           enqueue($Q_A$, nd)
123         else if isSymbolicBranch($\sigma$.nxt) then
124           $\pi \leftarrow \pi \,\|\, \langle\sigma.\mathsf{nxt}\rangle$
125           $\sigma'$ ← clone($\sigma$)
126           $\sigma'$ ← [execStep($\sigma'$) | $\sigma'$.nxt.cond ↦ false]
127           if $\sigma'$.cons $\not\Rightarrow$ false then
128             nd.child$_0$ ← makeNode($\pi$, $\sigma'$, nd.saved)
129             enqueue($Q_A$, nd.child$_0$)
130           $\sigma$ ← [execStep($\sigma$) | $\sigma$.nxt.cond ↦ true]
131           if $\sigma$.cons $\not\Rightarrow$ false then
132             nd.child$_1$ ← makeNode($\pi$, $\sigma$, nd.saved)
133             enqueue($Q_A$, nd.child$_1$)
```

Figure 8: VfyMsg procedure, described in Appendix A.1. Shaded lines implement the multipass algorithm and are described in Appendix A.2.

the newly found execution prefix Π_n; Rslt.state is set to the state that results from executing it, conditioned on the last message being msg_n (denoted [execStep(σ) | σ.nxt.msg ↦ msg_n]); and any prohibitive functions that were skipped are recorded for outputting assumptions (not shown, for notational simplicity). All other threads of execution now exit because Done = **true** and the parent procedure ParallelVerify will return Rslt.

In the final case, (isSymbolicBranch(σ.nxt)), the algorithm is at a symbolic branch. Thus, the branch condition contains symbolic variables and cannot be evaluated as true or false in isolation. Using symbolic execution, the algorithm evaluates both the true branch and the false branch by executing σ.nxt conditioned on the condition evaluating to **false** (denoted [execStep(σ') | σ'.nxt.cond ↦ **false**] in line 126) and conditioned on the branch condition evaluating to **true** (130). In each case, the constraints of the resulting state are checked for consistency (127, 131), for example, using an SMT solver.

If either state is consistent, it is atomically placed onto Q_A (129, 133).

A.2 The multipass algorithm

The multipass verification algorithm involves changes to the VfyMsg procedure in Fig. 8, specifically the insertion of the shaded lines. Whenever σ.nxt is a call to a prohibitive function, it is treated separately (lines 119–122), using the execStepProhibitive function (121). (To accomplish this, isNormal in line 106 now returns **false** not only for any network instruction or symbolic branch, but also for any call to a prohibitive function.) If execStepProhibitive receives a call σ.nxt to a prohibitive function with any symbolic input buffers, it replaces the call with an operation producing fully symbolic output buffers of the appropriate size. However, if the constraints saved in nd.saved allow the concrete input buffer values to be inferred, then execStepProhibitive instead performs the call σ.nxt on the now-concrete input buffers.

Prior to the execution path reaching a network instruction, when a call σ.nxt to a prohibitive function is encountered, nd.saved is simply **true** as initialized (not shown), permitting no additional inferences about the values of input buffers to σ.nxt. After a network instruction is reached and msg_n is reconciled with the constraints σ.cons accumulated along the path so far (110), the path constraints σ.cons and the new constraint σ.nxt.msg $= msg_n$ are saved in nd.saved (117). The execution path is then replayed from the root of the binary tree (i.e., beginning from Π_{n-1}, see 116). This process repeats until an execution occurs in which nothing new is learned (i.e., $(\sigma.\mathsf{cons} \wedge \sigma.\mathsf{nxt.msg} = msg_n) \equiv \mathsf{nd.saved}$, in 111), at which point VfyMsg returns as before.

B TLS Experimental Setup

In Sec. 6, we applied our client verification algorithm to OpenSSL, a widely used implementation of Transport Layer Security (TLS) with over 400,000 lines of code. In order to run OpenSSL symbolically in KLEE, some initial instrumentation (203 modified/added lines of code) was required: compiling to LLVM without x86 assembly, inserting symbolics at random number generators, and providing convenient record/playback functionality for testing the network SEND and RECV points. This manual one-time cost is likely unavoidable for symbolic execution of any application, but should be relatively small (0.05% of the OpenSSL codebase).

We then configured our OpenSSL client with one of the currently preferred cipher suites, namely `TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256`.

- Key exchange: Ephemeral Elliptic Curve Diffie-Hellman (ECDHE) signed using the Elliptic Curve Digital Signature Algorithm (ECDSA)
- Symmetric Encryption: 128-bit Advanced Encryption Standard (AES) in Galois/Counter Mode
- Pseudorandom function (PRF) built on SHA-256

Since our goal was to verify the TLS layer and not the higher-layer application, in our experiment we took advantage of the OpenSSL `s_client` test endpoint. This client establishes a fully functional TLS session, but allows arbitrary application-layer data to be sent and received via `stdin` and `stdout`, similar to the `netcat` tool. Verifying that network traffic is consistent with `s_client` is roughly equivalent to verifying the TLS layer alone.

The OpenSSL-specific user configuration for verification consisted of the following:

1. Configuring the following OpenSSL functions as prohibitive: `AES_encrypt`, `ECDH_compute_key`, `EC_POINT_point2oct`, `EC_KEY_generate_key`, `SHA1_Update`, `SHA1_Final`, `SHA256_Update`, `SHA256_Final`, `gcm_gmult_4bit`, `gcm_ghash_4bit`
2. Configuring `tls1_generate_master_secret` as the function to be replaced by server-side computation of the symmetric key.
3. (Optional) Declaring `EVP_PKEY_verify` to be a function that always returns success. This is a performance optimization only.
4. Configuring the whitelist of assumptions, containing one element to accept the client's ephemeral Diffie-Hellman public key.

The user configuration for OpenSSL, comprising declarations of prohibitive functions and their respective input/output annotations, consisted of 138 lines of C code using our API, which is implemented using C preprocessor macros. Fig. 9 shows an example prohibitive function declaration for the AES block cipher. In this macro, we declare the function signature, which comprises the 128-bit input buffer `in`, the 128-bit output buffer `out`, and the symmetric key data structure, `key`, which contains the expanded round keys. Both `in` and `key` are checked for symbolic data. If either buffer contains symbolic data, `out` is populated with unconstrained symbolic data, and the macro returns without executing any subsequent lines. Otherwise, the underlying (concrete) AES block cipher is called.

In a pure functional language or an ideal, strongly typed language, the prohibitive function specifications could in principle be generated automatically from the

```
DEFINE_MODEL(void, AES_encrypt,
             const unsigned char *in,
             unsigned char *out,
             const AES_KEY *key)
{
  SYMBOLIC_CHECK_AND_RETURN(
      in, 16,
      out, 16, "AESBlock");
  SYMBOLIC_CHECK_AND_RETURN(
      key, sizeof(AES_KEY),
      out, 16, "AESBlock");
  CALL_UNDERLYING(AES_encrypt,
      in, out, key);
}
```

Figure 9: Example prohibitive function declaration.

function name alone. Unfortunately, in C, the memory regions representing input and output may be accessible only through pointer dereferences and type casts. This is certainly true of OpenSSL (e.g., there is no guarantee that the AES_KEY struct does not contain pointers to auxiliary structs). Therefore, for each prohibitive function, the user annotation must explicitly define the data layout of the input and output.

The final configuration step involves specifying a whitelist of permitted assumptions. In the case of OpenSSL, only one is necessary, namely, that the Elliptic Curve Diffie-Hellman (ECDH) public key received from the client is indeed a member of the elliptic curve group (which, incidentally, the server is required to check [26]). In an actual verification run, the verifier produces an assumption of the form:

```
ARRAY(65) ECpointX_0___ECpoint2oct_0 =
  0x04001cfee250f62053f7ea555ce3d8...
```

This assumption can be interpreted as the following statement: the verifier assumes it is possible for the client to create a buffer of length 65 bytes with contents 0x04001cf... by first making the zeroth call to an ECpoint generation function, and then passing its output through the zeroth call to an ECpoint2oct (serialization) function. The single-element whitelist for OpenSSL is therefore simply:

```
ARRAY(*) ECpointX_*___ECpoint2oct_*
```

The asterisks (*) are wildcard symbols; the ECDH public key can be of variable size, and for all n, m, we permit the nth call to ECpointX and the mth call to the ECpoint2oct functions to be used as part of an entry in the whitelist.

The domain knowledge required for the first two configuration steps is minimal, namely that current TLS configurations use the above cryptographic primitives in some way, and that a symmetric key is generated in a particular function. The domain knowledge necessary for the third configuration step is that TLS typically uses public key signatures only to authenticate the server to the client, e.g., via the Web PKI. The server itself generates the signature that can be verified via PKI, and so the verifier knows that the chain of signature verifications is guaranteed to succeed. Moreover, this optimization generalizes to any protocol that uses a PKI to authenticate the server to an untrusted client. The domain knowledge necessary for the fourth configuration step is non-trivial; whitelisting a cryptographic assumption can have subtle but significant effects on the guarantees of the behavioral verifier (e.g., off-curve attacks). However, the number of assumptions is small: for OpenSSL, there is only the single assumption above.

C Lazy Constraint Generators

There are a number of cases in which a behavioral verifier requires an extra feature, called a *lazy constraint generator*, to accompany the designation of a prohibitive function.

Since a function, once specified as prohibitive, will be skipped by the verifier until its inputs are inferred concretely, the verifier cannot gather constraints relating the input and output buffers of that function until the inputs can be inferred via other constraints. There are cases, however, where introducing constraints relating the input and output buffers once some *other* subset of them (e.g., the output buffers) are inferred concretely would be useful or, indeed, is central to eventually inferring other prohibitive functions' inputs concretely.

Perhaps the most straightforward example arises in symmetric encryption modes that require the inversion of a block cipher in order to decrypt a ciphertext (e.g., CBC mode). Upon reaching the client SEND instruction for a message, the verifier reconciles the observed client-to-server message msg_n with the constraints σ.cons accumulated on the path to that SEND; for example, suppose this makes concrete the buffers corresponding to outputs of the encryption routine. However, because the block cipher was prohibitive and so skipped, constraints relating the input buffers to those output buffers were not recorded, and so the input buffers remain unconstrained by the (now concrete) output buffers. Moreover, a second pass of the client execution will not add additional constraints on those input buffers, meaning they will remain unconstrained after another pass.

We implemented a feature to address this situation by permitting the user to specify a lazy constraint generator along with designating the block cipher as prohibitive. The lazy constraint generator takes as input some subset of a prohibitive function's input and output buffers, and produces as output a list of constraints on the prohibitive function's other buffers. The generator is "lazy" in that it will be invoked by the verifier only after its inputs are inferred concretely by other means; once invoked, it produces new constraints as a function of those values. In the case of the block cipher, the most natural constraint generator is the inverse function, which takes in the key and a ciphertext and produces the corresponding plaintext to constrain the value of the input buffer.

More precisely, a lazy constraint generator can be defined as a triple $L = (inE, outE, f)$ as follows:

1. *inE*: A set of symbolic expressions corresponding to the "input" of f.
2. *outE*: A set of symbolic expressions corresponding to the "output" of f.
3. f: A pure function relating *inE* and *outE* such that if f could be symbolically executed, then the constraint $f(inE) = outE$ would be generated.

The set of expressions *outE* corresponds to the output of f, not to the output of the lazy constraint generator. The lazy constraint generator L is blocked until the set of expressions *inE* can be inferred to uniquely take on a set of concrete values (e.g., $inE = 42$). At that point, L is "triggered", generating the real constraint $outE = f(inE)$ that can be added to the path condition (e.g., $outE = f(42) = 2187$). Note that f may correspond to a prohibitive function $p(x)$ or it may correspond to its inverse $p^{-1}(x)$. In the latter case, the expressions *inE* represent the "input" to f but represent the "output" of a prohibitive function $p(x)$.

In order to create a lazy constraint generator, a prohibitive function definition (i.e., DEFINE_MODEL...) includes an additional call to the special function:

```
DEF_LAZY(uint8 *inE, size_t inE_len,
         uint8 *outE, size_t outE_len,
         const char *f_name)
```

The f_name parameter designates to the verifier which function should be triggered if inE can be inferred concretely. Note that since there is only one input buffer and one output buffer, some serialization may be required in order to use this interface, but it is straightforward to wrap any trigger function appropriately.

We illustrate lazy constraint functionality using two example clients. First, let p and p_inv be the following "prohibitive" function and its inverse.

```
unsigned int p(unsigned int x) {
    return 641 * x;
}

unsigned int p_inv(unsigned int x) {
    return 6700417 * x;
}
```

The correctness of the inversion for $p(x)$ can be seen from the fact that the fifth Fermat number, $F_5 = 2^{32} + 1$, is composite with factorization $641 \cdot 6700417$. Since addition and multiplication of 32-bit unsigned integer values is equivalent to arithmetic modulo 2^{32}, we have:

$$\begin{aligned} p^{-1}(p(x)) &= 6700417 \cdot 641 \cdot x \\ &= \left(2^{32} + 1\right) \cdot x \\ &\equiv (1) \cdot x \pmod{2^{32}}. \end{aligned}$$

In place of $p(x)$, one could use any function with a well-defined inverse, such as a CBC-mode block cipher.

Fig. 10 shows an example program where p is a prohibitive function. Assume that the lazy constraint generator represents p^{-1}, the inverse function. Suppose that when execution reaches SEND(y), the corresponding data observed over the network is 6410. This implies a unique, concrete value for y, so the lazy constraint generator is triggered, generating a new constraint, $x = p^{-1}(6410) = 10$. Any observed value for SEND(x) other than 10 causes a contradiction.

Fig. 11 shows a slightly trickier test case. Here, assume that the lazy constraint generator is defined to correspond to the original function $p(x)$, instead of the inverse. Along the particular branch containing SEND(y), the concretization of variable x is implied by the path condition (x == 10) rather than by reconciling symbolic expressions at a network SEND point. The implied value concretization (IVC) of x triggers the execution of $p(x) = p(10) = 6410$ and results in the new constraint $y = 6410$. This constraint will then be matched against the network traffic corresponding to SEND(y). Note that at the time of writing, KLEE did not provide full IVC. As a workaround, we inserted extra solver invocations at each SEND to determine whether the conjunction of the path condition and network traffic yielded enough information to trigger any lazy constraint generators that have accumulated.

```
int main() {
    unsigned int x, y;
    MAKE_SYMBOLIC(&x);
    y = p(x);
    SEND(y);
    SEND(x);
    return 0;
}

// Positive test case: 6410, 10
// Negative test case: 6410, 11
```

Figure 10: Client for which a lazy constraint generator could be triggered due to reconciliation with network traffic at a SEND point.

```
int main() {
    unsigned int x, y;
    MAKE_SYMBOLIC(&x);
    y = p(x);
    SEND(314);
    if (x == 10) {
        SEND(y);
    } else {
        SEND(159);
    }
    return 0;
}

// Positive test case: 314, 159
// Positive test case: 314, 6410
// Negative test case: 314, 6411
```

Figure 11: Client for which a lazy constraint generator could be triggered due to the implied value at a symbolic branch.

Note that our OpenSSL case study in Sec. 5.4 does not require lazy constraint functionality since in the AES-GCM encryption mode, the ciphertext and plaintext buffers are related by simple exclusive-or against outputs from the (still prohibitive) block cipher applied to values that can be inferred concretely from the message. So, once the inputs to the block cipher are inferred by the verifier, the block cipher outputs can be produced concretely, and the plaintext then inferred from the concrete ciphertexts by exclusive-or.

Although the CBC-mode cipher suites in TLS are no longer preferred due to padding-oracle attacks [33], a number of legacy clients still run them, as they lack support for the newer Authenticated Encryption with Associated Data (AEAD) modes such as AES-GCM. Lazy constraint generation enables behavioral verification of these legacy clients.

In the limit, lazy constraint generators could be configured for every single prohibitive function, such that triggering one could cascade into triggering others. This would essentially mimic the functionality of the multi-pass algorithm of Sec. 5. An advantage of this approach would be that the “subsequent passes” (as emulated by cascading lazy constraints) execute only the code that was skipped via the prohibitive function mechanism—other unrelated code does not require re-execution. The tradeoff is an increase in the number of solver queries used to (1) detect that lazy constraints can be triggered, and to (2) stitch the new constraints into the path condition and check whether satisfiability is maintained. Future work could compare the performance tradeoffs between these two mechanisms.