Discovering Specification Violations in Networked Software Systems

Robert J. Walls[⋆] Yuriy Brun[UM] Marc Liberatore[UM] Brian Neil Levine[UM]
[⋆]Computer Science and Engineering
Pennsylvania State University
University Park, PA
rjwalls@cse.psu.edu
[UM]College of Information and Computer Sciences
University of Massachusetts, Amherst
Amherst, MA
{brun, liberato, brian}@cs.umass.edu

***Abstract*—Publicly released software implementations of network protocols often have bugs that arise from latent specification violations. We present APE, a technique that explores program behavior to identify potential specification violations. APE overcomes the challenge of exploring the large space of behavior by dynamically inferring precise models of behavior, stimulating unobserved behavior likely to lead to violations, and refining the behavioral models with the new, stimulated behavior. APE can (1) discover new specification violations, (2) verify that violations are removed, (3) identify related violations in other versions and implementations of the protocols, and (4) generate tests. APE works on binaries and requires a lightweight description of the protocol's network messages and a violation characteristic. We use APE to rediscover the known heartbleed bug in OpenSSL, and discover one unknown bug and two unexpected uses of three popular BitTorrent clients. Manual inspection of APE-produced artifacts reveals four additional, previously unknown specification violations in OpenSSL and μTorrent.**

I. Introduction

Despite significant effort and resources spent on ensuring software quality, software systems often ship with bugs and security vulnerabilities. These vulnerabilities enable cybercrime, which in 2010 cost $114 billion globally [45] and has affected one-third of U.S. households [19]. Finding violations against a software specification and fixing security bugs is expensive, time consuming, and difficult [3], [12], [44]. Discovering differences between a protocol specification and implementations is especially hard in networked software. Networked communication introduces an inherent nondeterminism, error states are difficult to reproduce, and undefined or unspecified behavior is often not handled consistently by different implementations.

This paper describes and evaluates APE, a technique for testing and detection of *specification violations* in networked software. Such violations can result in unexpected behavior, incompatibilities, bugs, and vulnerabilities.

APE can be used in four ways: First, APE can apply limited human insight to explore system behavior and discover specification violations in system implementations. Second, given a patch for an exploit, APE can verify the patch and provide evidence that there are no other, similar execution paths that trigger the same exploit. Third, APE can apply exploits known for some implementation of a protocol to other implementations, and tweak the exploits to work on new implementations. This process is particularly useful when updates only partially fix a vulnerability. Fourth, APE can generate tests of previously untested behavior. Overall, APE helps find specification violations, adapt existing exploits, and test implementations against existing exploits.

APE uses a form of network-based fuzz testing to observe and explore a *target system*'s behavior, and infers a precise state-based model of that behavior. Using the model, APE identifies unexplored behavior, stimulates the system to execute that behavior, and iteratively refines the model using the new observations. Then, APE uses the model and a user-specified description of a specification violation (e.g., "download a file without contributing any uploaded data") to propose a set of potential candidate executions that satisfy the violation description. Depending on the precision of the inferred model, some of these candidate executions may correspond to real system executions, while others may not. APE verifies the candidate violation executions using the target implementation.

The paper's main contributions are:

- APE, a technique for discovering specification violations in networked software. APE works on compiled binaries and does not require access to the target system's source code. Instead, the user needs to provide executable methods for sending and receiving network messages, and a description that identifies when a specification has been violated.
- The process for using APE to (1) verify a patch, (2) apply known exploits to a multitude of implementations, and (3) tweak exploits to discover related but distinct exploits.
- A case study applying APE to two versions of OpenSSL to (1) discover and reproduce the heartbleed vulnerability in one version, (2) verify it was patched in the other, and (3) discover two additional specification violations.
- A case study applying APE to three popular implementations of BitTorrent: μTorrent, Transmission, and Azureus, discovering five bugs.
- An open-source, prototype APE implementation and all code used in our evaluations: `http://forensics.umass.edu/ape.php`.

APE builds on prior automated behavior exploration and test generation work (e.g., TAUTOKO [22], ProCrawl [43], and MACE [16]) by targeting networked systems, and expanding violation expressiveness by allowing modifying messages, as opposed to focusing exclusively on method call sequences.

The rest of this paper is structured as follows. Section II describes APE using a simple example, and Section III details APE design and implementation. Section IV evaluates our approach's utility in discovering specification violations on OpenSSL, and Section V evaluates that utility on BitTorrent. Section VI evaluates APE's efficiency. Section VII places our work in the context of related research. Finally, Section VIII summarizes our contributions.

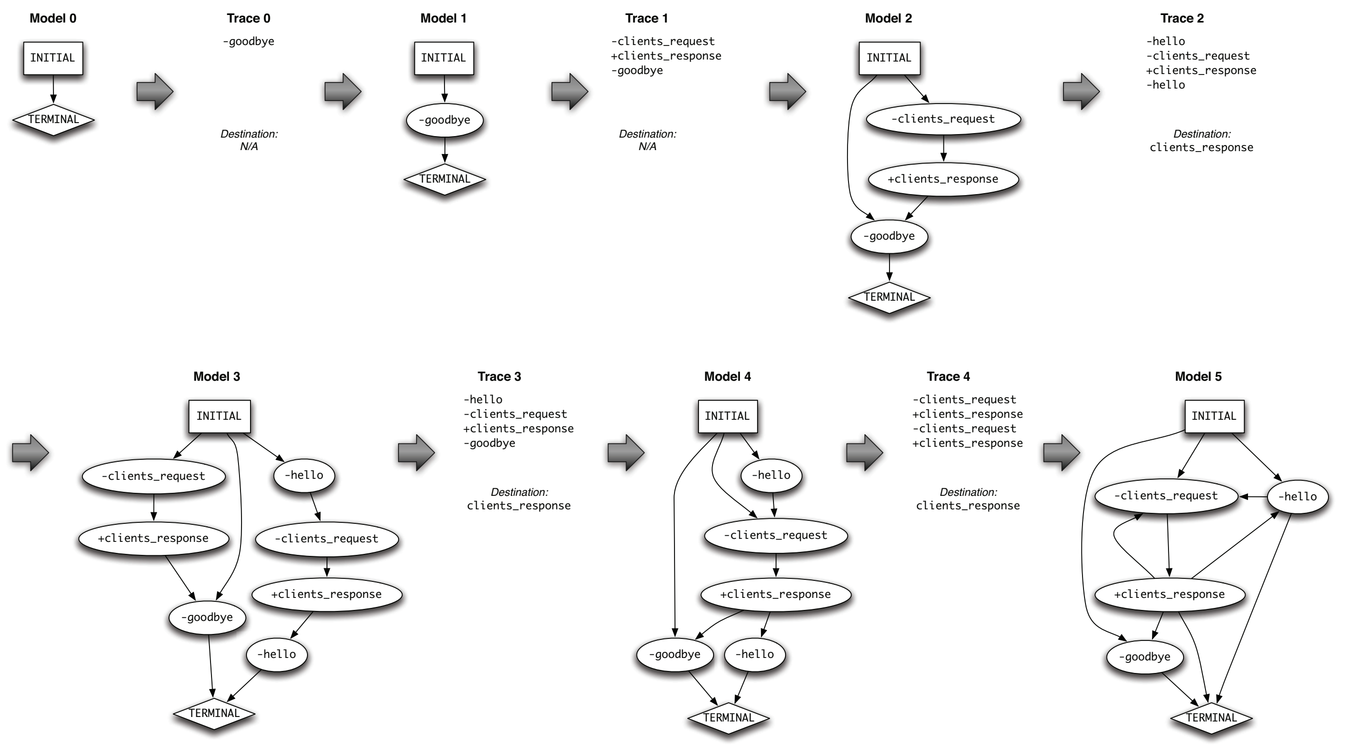

Fig. 1. APE uses models of its target's (here TRACKER's) behavior to guide exploration and generate traces of new executions. For example, using Model 3, APE explores an execution that generates Trace 3, which APE then uses (together with Traces 0, 1, and 2) to build Model 4. Messages *to* the target are prefixed with -, and messages *from* the target are prefixed with +.

II. MOTIVATING EXAMPLE

APE's goal is to locate and exploit specification violations in networked software. The key idea behind APE is to dynamically infer a model of a target software system behavior (using recent advances in automated behavioral model inference [9]) and use that model to guide new executions toward potential violations of the specifications. APE then uses these newly observed executions to improve the inferred model, and uses the improved models to better guide future executions, iterating this guided behavior exploration to find violations. To illustrate APE, we first develop a running example using the TRACKER protocol. TRACKER is a simple client-server protocol we built to keep track of clients connected to a server.

The TRACKER protocol specification. Clients join a network and identify themselves to the TRACKER server using a `hello` message, at which point, the server verifies the client has permission to join the network. Whenever a client sends a `clients_request` message to the server, the server responds with a `clients_response` message with a list of all clients that have connected since the previous update. Clients leave the network by sending a `goodbye` message.

Example TRACKER implementation. Consider the following code, part of our Python TRACKER server implementation.

```
1 def handle_message(IP, message):
2     if message == "hello" and \
3         self.is_authorized(IP):
4           self.connected_peers.append(IP)
5           self.new_peers.append(IP)
6     elif message == "clients_request":
7         self.handle_clients_request(IP)
8     elif message == "goodbye":
9         self.connected_peers.remove(IP)
```

The `handle_message(...)` method determines the server's response to clients' messages. This method contains two bugs that allow a client to violate the intended operation of the protocol. First, lines 6–7 fail to check if the client is authorized before processing the `clients_request` message. Second, an already connected client can trick the server into thinking it is a new peer and cause the server to erroneously broadcast the peer's presence with the next `clients_response` message. This happens because the server fails to check if the client has already connected when it handles the `hello` message in lines 2–5, and appends the IP to the list of new peers in line 5. We focus on the first bug.

Applying APE to TRACKER. APE can discover these specification violations with knowledge of only the allowed messages and without access to the source code. Here, we give a high-level overview of how APE does so.

We refer to the implementation in which APE is attempting to discover bugs as the *target*. APE discovers specification violations through interaction with the target by observing how the target reacts to different messages and using these observation to learn and explore the target's behavior.

APE has three stages of operation, described here in the context of TRACKER: exploration, candidate violation discovery, and violation verification. The descriptions in this section are high-level to build intuition; Section III describes the details.

During the exploration stage, APE learns and refines a model of the target's behavior by stimulating the target by sending it sequences of messages and observing its behavior. APE systematically chooses which messages to send by using a model of what it already knows about the allowed behavior, and by perturbing previously attempted executions. This results in a form of fuzz testing. Whenever an execution completes, APE restarts the client and begins a new execution with a different sequence of messages. APE logs all messages that are sent and received during the executions and uses this log to refine the model of known behavior.

APE uses *Synoptic* [9] to convert observed execution traces into a finite state machine (FSM) model of the target's behavior. (While APE could instead use other model-inference algorithms [6], [7], [8], [20], [23], [25], [32], [37], [39], [40], [41], [42], [43], [48], our experience showed that Synoptic model's enforcement of observed temporal invariants leads to sufficiently precise models for APE's purposes.) Each *path* through the FSM model represents an execution, in terms of the sequence of messages sent and received by APE. Synoptic models are predictive: they describe all observed executions *and* predict unobserved executions that satisfy key temporal invariants mined from the observed executions.

APE uses the inferred models to guide its behavioral exploration. APE executes the target, following the previously-unobserved, predicted paths, probabilistically mutating them to create more diversity in the exploration. APE refines the model of the target's behavior by iteratively executing potentially mutated paths, collecting execution traces, and inferring models based on the observed executions. This model-based exploration is better targeted and more efficient than the alternative fuzz testing approach of sending random sequences of network messages and observing the target's behavior.

Figure 1 illustrates five iterations of the exploration process. Model 0 shows the starting FSM model, which encodes no behavior and has just two connected states labeled `INITIAL` and `TERMINAL`. The first round of exploration generates a single trace (Trace 0): sending a `goodbye` message (in the traces, sending is denoted by a `-`, and receiving by a `+`) generates no response. APE improves its understanding of TRACKER by using Synoptic to infer Model 1 from Trace 0. Next, using Model 1, APE picks a path (sending a `goodbye`) and permutes it slightly to generate a new Trace 1: Sending a `clients_request` ahead of the `goodbye` message results in receiving a `clients_response` message. Inferring an FSM model from the two observed traces together results in Model 2. Repeating this process further refines the model of TRACKER's behavior. Note that Model 5 includes not only observed behavior, but also predicted behavior, such as the trace: ⟨`-client_request`, `+client_response`, `-hello`, `-client_request`, `+client_response`, `-goodbye`⟩. These predicted paths allow APE to explore the target's behavior more efficiently than by guessing randomly or through exhaustive testing.

APE's second stage, candidate violation discovery, analyzes the model for potential specification violations. To run APE, the user specifies a vulnerability characteristic (see Section III-A). For example, if a user is interested in finding if an implementation of TRACKER client can ever receive a `clients_response` without having sent a `hello`, the user specifies the vulnerability characteristic "`clients_response` should always be preceded by `hello`". Model 5 in Figure 1 includes one path that violates the characteristic: ⟨`INITIAL`, `-clients_request`, `+clients_response`⟩. APE searches the

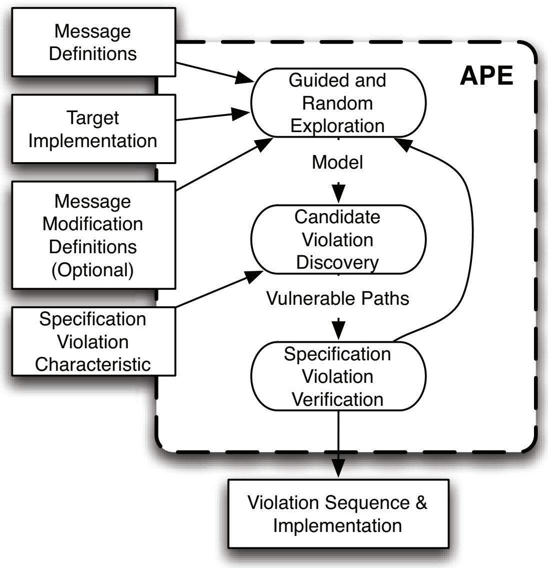
Message Definitions
Target Implementation
Message Modification Definitions (Optional)
Specification Violation Characteristic
APE
Guided and Random Exploration
Model
Candidate Violation Discovery
Vulnerable Paths
Specification Violation Verification
Violation Sequence & Implementation

Fig. 2. A high-level overview of APE and its three stages: exploration, analysis, and verification.

model to find all such loop-free paths.

Whenever APE finds candidate violation paths, its third stage, violation verification, executes the candidates against the target implementation. As some paths in the model are predicted, rather than observed, this verification step ensures the candidates are real. APE can report verified violation exploits to the user, and generate an implementation of the tester that follows that path of execution to exploit the violation.

APE can both discover specification violations and generate variations of known exploits to work on new implementations. To find new exploits, APE starts its guided exploration with an empty FSM model. To modify existing exploits, APE starts with an FSM model of behavior that includes those exploits, and guides the exploration along those exploits. Additionally, if APE starts its exploration with test suite executions, it will explore the target's behavior to first generate and then verify new executions untested by the test suite.

III. APE SYSTEM DESIGN

Figure 2 shows the process APE follows to find specification violations in software implementing a protocol. APE has three required and one optional inputs: (1) a binary of the target software system implementation, (2) a description of the messages the system may send as part of its protocol, (3) a set of descriptions for identifying candidate violations, and (4) optionally, a set of descriptions for modifying legal messages. APE does not require access to the target system source code. APE operates in three stages. First, APE *explores* the behavior of the target system. Second, APE *analyzes* the model to find likely system executions that fit the violation descriptions. Third, APE *verifies* that the executions are real violations. We now use the TRACKER example from Section II to describe the inputs and the three stages.

A. APE *Inputs*

Target implementation binary. APE uses a binary of the target system and needs to be able to start and stop its execution.

```
class TesterProtocol(ExploreProtocol):
  def send_hello(self):
    self.transport.write(
      struct.pack('!I', 5) + 'hello')

  def send_clients_request(self):
    self.transport.write(
      struct.pack('!I', 15) +
        'clients_request')

  def send_goodbye(self):
    self.transport.write(
      struct.pack('!I', 7) + 'goodbye')

  def handle_clients_response(self, data):
    pass

  def get_message_type(self, message):
    #Use to determine message type
```

Fig. 3. A TRACKER message types description written in Python has three methods for sending messages, one method for receiving a message, and one method to parse received messages into their event types. The `struct.pack` is a standard python function, and `transport.write` is provided by the Twisted [46] networking framework.

For TRACKER, this means the command to start and to kill the TRACKER server process.

Message definitions. APE also requires a description of the messages that can be sent to and received from the target. This descriptions consists of executable methods for sending and receiving messages to and from the target. There is one method for each message that can be sent, a single method that receives and parses messages, and a method that returns a message's event type.

Figure 3 shows the five-method message description for TRACKER: three for sending the different message types; one for handling the received `clients_response` message; and one for translating the received payloads into event types. For TRACKER, this last method is trivial because we can only receive a single type of message, `clients_response`. To test a more complicated system, such as BitTorrent (described in Section V), a more complex set of methods may be required. Our BitTorrent executable message description is structured similarly to the description for TRACKER, but includes the logic and state necessary to minimally participate in the BitTorrent protocol.

Specification violation characteristic. As a final required input, APE needs a way to recognize when executions exploit a specification violation. APE uses the violations description to identify which paths in the model have the potential to exploit the violation. We delay our detailed description of this input until Section III-C.

Message modifications. Optionally, APE accepts a description of how messages may be modified. Without such a description, APE can discover specification violations by altering the *order* of legal network messages. With this optional input, APE can also discover violations by altering the *content* (often called *payload*) of the messages. In our implementation, this input consists of several executable methods: one for each sent message that can be modified that describes how to generate such messages.

We write methods that generate varied message content by specifying the fields of the message, which fields can be modified, and how the fields can be modified. For example, Section IV describes our OpenSSL `heartbeat` [28] message generator. The `payload` field is a random-sized series of random bits. And the `payload_length` field is chosen uniformly at random to be either the size of the payload field, a random integer larger than the size of the payload field, or a random integer smaller than the size of the payload field. Specifying a finite number of values from which each field can be selected randomly is another approach that could be implemented.

B. Exploration

The goal of the exploration stage is to interact with the target implementation and learn how it responds to sequences of messages. The product of this stage is an FSM model of the implementation's behavior. The FSM has a state for each message that can be sent or received, and two special `INITIAL` and `TERMINAL` states; the paths from `INITIAL` to `TERMINAL` represent potential executions.

The exploration stage executes the target implementation under varying conditions and collects traces of the network messages sent and received during execution. APE then uses Synoptic [9], a model-inference technique, to automatically infer an FSM from those execution traces. The FSM model is a generalization of the observed system behavior; it includes paths that represent all the observed behavior, but it also includes more paths that the model-inference algorithm deemed similar enough to be likely possible (although unobserved) executions.

APE uses guided exploration (Section III-B3) to generate execution traces more efficiently than via random exporation. It starts with known behavior and perturbs it slightly to increase the chances that attempted executions are successful. The goal is to add meaningful information to its knowledge of the target's behavior with each generated trace. However, when APE starts with no knowledge about the behavior, it is forced to use random exploration (Section III-B1) until it learns a little behavior and can switch to guided exploration.

1) Random exploration: The base case for APE is to have no knowledge of a target implementation's behavior. This case is represented by a model with only two states, `INITIAL` and `TERMINAL`, as shown in Model 0 of Figure 1. In this case, there is no known behavior to perturb, so APE explores randomly.

During random exploration, APE uses the model to infer the most likely state of the target system, and sends messages that have not been previously sent to the system in that state. To determine its current model state, APE examines the last k received messages. APE finds all instances of this k-sequence as a path in the model and reasons that the current state must be at the end of one of these sequences. It then chooses a message uniformly at random from the set of messages not yet sent from the set of current possible states. If that set of messages is empty, APE selects a message uniformly at random from among all messages. If APE fails to find any k-paths, and thus any candidate states, it repeats the process using a value of $k-1$. If there are no valid candidate states for $k=1$ then APE selects a message to send uniformly at random from among all messages. In practice, we found that $k=5$ provided an adequate balance between performance and accuracy.

Example. Recall that the two-state initial model for TRACKER has no received message events, so APE starts Trace 0 (Figure 1) by sending a single random message, in this example, a `goodbye` message. As might be expected, the TRACKER server does not reply to the `goodbye` message and the trace terminates after a timeout. APE then uses Synoptic (Section III-B2) to infer Model 1 from Trace 0. Model 1 contains just a single path from `INITIAL` to `TERMINAL` via `-goodbye`.

Model 1 still has no receive events, so APE again enters random exploration mode. This time, it sends a `clients_request` message, waits (using a timeout) for a possible response, receives a `clients_response` message, and then sends a `goodbye` message (Trace 1). APE now uses Synoptic to infer Model 2 from Traces 0 and 1. Now that the model has a receive event, `clients_response`, it will use guided exploration. We next describe model inference, and then guided exploration.

2) FSM Model Inference: APE uses the observed execution traces to infer a predictive FSM model. There are many existing techniques for such model inference [6], [8], [9], [20], [23], [25], [37], [39], [41], [42], [43], [48], and it is not the focus of this paper to improve on them. Instead, APE uses Synoptic because of its precise predictive properties and previous use for manual software debugging [9].

Synoptic infers a model by mining a set of temporal invariants present in the observed execution, such as `hello` always eventually followed by `goodbye`. Synoptic then builds a concise model of the observed traces, and uses model checking and counterexample-guided abstraction refinement (CEGAR) [17] to eliminate predicted paths that do not satisfy the mined temporal invariants. The end result is a precise and concise model that includes all observed executions and predicts unobserved executions deemed likely because they satisfy the mined temporal invariants.

Models represent sending and receiving messages as events. When all messages of a given type are identical, this abstraction is straightforward. However, when messages can be modified (via the user-specified message modification methods), APE allows for two ways to represent these messages in the model: (1) APE can create a unique name for each modified message instance. This approach's main drawback is that Synoptic's predictive power is reduced, which can affect the efficiency of APE's exploration. However, the primary benefit is that once APE finds a candidate violation in a model (Section III-C), the model contains sufficient information to verify it (Section III-D). (2) APE can abstract all modified messages with a single message name. This approach enables Synoptic's predictive power, but requires, once a candidate violation path is identified, reanalyzing the execution traces to identify which executions led to the path, and how to recreate the executions' modified messages. While APE supports both approaches, the evaluation in Section IV follows the former.

Throughout exploration, APE periodically updates its FSM model using all of the traces collected up to that point. The frequency of the model updating is configurable; for exposition, for the TRACKER example, APE updates the model after every execution trace.

3) Guided exploration: APE's guided exploration starts with an FSM model and explores along the paths already in the model, but introduces deviations from those paths to discover new behavior. Because Synoptic's models are predictive — they include likely possible but unobserved behavior — both paths in the model and deviations from those paths can produce evidence of new, unobserved behavior.

Choosing a destination. To guide its exploration, APE first selects a *destination event*, uniformly at random, from among all message types that appear as *received* messages in the model. In other words, APE's goal is to coerce the target into responding with a specific message. For TRACKER, `clients_response` is the only receivable message, and so it is the only possible destination.

To avoid falling into local minima, with a low probability, APE forgoes guided exploration and enters random exploration mode. This probability is inversely proportional to the number of distinct event labels in the current model.

Choosing a path. After choosing a destination, APE randomly picks one of the paths from `INITIAL` to one of the states that represent the selected received message type. For example, in Model 3 in Figure 1, there are two possible paths to states labeled with `+clients_response`: (1) `-clients_request` and (2) `-hello`, `-clients_request`. APE's guided exploration's goal is to guide the target to the destination state by inducing an execution along the chosen path (possibly with some deviations).

Sending messages. An execution consists of a sequence of sets of sent events and received events. For example, in Trace 2, APE sends `hello` and `clients_request` messages, receives a `clients_response`, and sends a `hello`. Using a path, APE identifies the first set of messages it needs to send to follow that path. For each message in that set, APE may perturb the message with a small probability. APE may randomly skip the message entirely. If the message is not skipped, then APE may substitute a different message, chosen uniformly at random from the set of all possible send messages. In practice, for the systems and models we have used, we find that a probability of $1/11$ works well for both skipping and substitution. Once APE chooses and sends the message, it waits up to 0.5 seconds for the target's response. If there is no unexpected response, APE moves on to the next set of send messages in the path, following the same perturbation procedure until it has traversed the entire path and received the destination message. APE then switches to random exploration mode to finish the current execution and generate a new trace.

Determining the current model state. APE may not always receive the expected set of messages on the path it is following. This outcome can happen because APE perturbed the path, or because the path was predicted inaccurately by Synoptic. Whenever APE receives an unexpected message, it attempts to reconcile that message with the current model. To do this, APE tries to determine its current state in the model using the last *k* received messages, just as it did in its random exploration mode (recall Section III-B1). If that process cannot find a suitable state, APE switches into random exploration mode to finish the current execution and generate a new trace.

Naturally, exploring system behavior in this way generates new, previously unobserved executions. Starting APE from a set of test case executions will lead it to generate new, behaviorally-distinct tests.

Example. Since Model 3 contains received events, APE picks `clients_response` as the destination event for the new execution (Trace 3). It selects the `-hello`, `-clients_request`, `+clients_response` path and sends the first two messages, by chance, without perturbing them. APE receives the expected `+clients_response` and finally switches to random exploration mode, choosing to send a `-hello` message. The target then closes the connection. APE then uses Synoptic to infer Model 4 from Traces 0–3.

4) Exploration with message modifications: If APE is given the optional description of how message content may be modified, APE uses a two-phase approach to exploration. First, it uses standard guided exploration without modifying any messages to build an initial model of the target system's behavior. Next, every time it sends a message of a particular type, it uses the message modification method to generate a new message of that type and send it. (The message modification method may choose, of course, to generate an unmodified message.)

C. Candidate Violation Discovery

After a user-specified number of trace generation and model refinement iterations, APE takes the current model and searches it for specification violations. APE uses the violation characteristic (recall Section III-A) to identify candidate violations.

In practice, Ape can be execute the exploration and analysis stages in parallel, continually refining the model while checking properties in previous models.

Specification violation characteristic. A violation characteristic is a function whose input is an FSM model and output is a set of paths through the model. The intent is for the paths to expose a violation. For example, for TRACKER, this characteristic could be a function that returns all paths that allow a peer to receive new peer information without authenticating, which exposes the first bug in our TRACKER implementation (recall Section II). However, to simplify the use of APE, we allow for a highly simplified definition of the violation characteristic: The user needs only to describe two conditions, (1) an ordered sequence of states that must be present in the path, not necessarily contiguously, and (2) a set of states that must not be on the path. For example, for TRACKER, the characteristic specifies that the violation-exposing path contains `INITIAL` and `clients_response`, but does not contain `hello`.

Enumerating candidate paths. Given a model and a set of candidate violation characteristics, APE finds candidate paths using a straightforward graph search algorithm. The current APE implementation removes the set of states that must never occur on a candidate path from the model, along with all edges to or from those states. Then, APE uses depth-first search to find all (loop-free[1]) paths between sequential pairs of states that must occur in the violation-exposing path. Finally, APE combines the paths between the sequential pairs of states to produce complete candidate violation-exposing paths.

For example, for TRACKER, APE uses Model 5 from Figure 1 to remove the `-hello` state and adjacent edges from the model and then find the only loop-free path that contains `INITIAL` and `-clients_response`.

D. Specification Violation Verification

Finally, APE uses the target system binary to verify the candidate violation-exposing paths. It follows the path to interact with the binary, much in the same way as it does during exploration except without perturbing the path. For example, in Section III-C, APE identified `INITIAL`, `-clients_request`, `+clients_response` as a candidate violation-exposing path, so it will start TRACKER, send a `clients_request` message, and then wait to receive the `clients_response` message. In this example, the buggy TRACKER server will respond with `clients_response`, verifying the violation.

[1]Cycles produce infinitely many paths, so we have to choose some limit, e.g., traversing loops a fixed number of times (also a common approach for symbolic execution). Additionally, because loops start and end at the same state, traversing the loop has no effect on protocol state from the perspective of the inferred model.

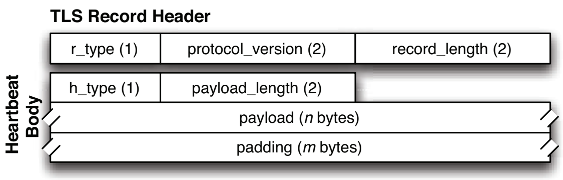

Fig. 4. The layout of a TLS heartbeat message.

APE tries multiple times to verify the path, in case of nondeterminism. If after verification, APE fails to produce at least one verified violation-exposing path, it returns to the exploration stage. While APE uses violation verification to test automatically discovered candidate violations, it can just as well test other candidates. For example, it could test a violation that was present in an earlier version of an implementation, or in other implementations of the same protocol. By using exploration, APE can start with a specification violation from another version or implementation, and explore if the target implementation is vulnerable to variants of those violations.

IV. EVALUATION: OPENSSL

This section details the application of APE to OpenSSL's implementation of the TLS Heartbeat Extension Protocol. APE was effective in discovering a known buffer-overflow vulnerability (the heartbleed vulnerability) and verifying that another OpenSSL implementation had the vulnerability patched. Further, manual inspection of APE-inferred behavioral model discovered two additional specification violations in both implementations. We have reported these bugs to the developers.

A. The Heartbleed Bug

Last year, researchers disclosed a bug in OpenSSL's implementation of the heartbeat extension protocol for TLS that allows an attacker unauthorized access to private keys [28]. The heartbeat extension protocol (defined in RFC 6520, https://tools.ietf.org/html/rfc6520) provides a mechanism for clients to maintain a connection to the server by sending a heartbeat message, requesting a response that echoes the heartbeat. The so-called *heartbleed* bug allows an attacker to specify a small payload but request a larger payload to be echoed back. The vulnerable OpenSSL servers, without checking the bound condition, respond with internal, private memory state.

B. Testing the Heartbeat Protocol

We applied APE to the TLS heartbeat protocol on two different Ubuntu 12.04 servers running Apache with OpenSSL. One server was vulnerable to the heartbleed bug, and the other was patched. APE used the payload modification option described in Section III-A to test these implementations.

Figure 4 shows the layout of a heartbeat request message. The heartbeat protocol is layered on top of the TLS record protocol. The record header (the TLS protocol refers to messages as records) consists of three fields making up 5 bytes: `record_type`, `protocol_version`, and `record_length`. A `record_type` of `0x15` denotes a heartbeat message. The `record_length` is a 2-byte integer specifying the length of the remaining fields of the heartbeat message: `heartbeat_type`, `payload_length`, an arbitrary `payload`, and `padding`. The

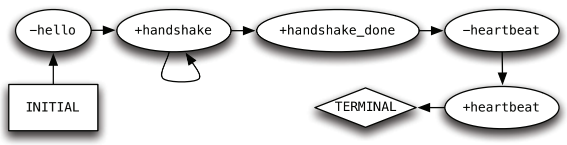

Fig. 5. APE-inferred model without message modification. After inferring this model, APE then explored the effects of modifying the heartbeat message.

padding is random data intended to be ignored by the server. To test this protocol, a developer needs to provide APE with a violation characteristic and the methods needed to generate the aforementioned fields. Both of these inputs can be derived directly from information in the protocol specification.

We configured APE by specifying the message modification method (recall Section III-A) to modify four of these fields: `record_length`, `payload_length`, `payload`, and `padding`. The `payload` and `padding` fields are each a randomly sized series of random bits. The `payload_length` field is uniformly randomly chosen to be either the size of the payload field, a random integer larger than the size of the `payload` field, or a random integer smaller than the size of the `payload` field. Similarly, the `record_length` field is uniformly randomly chosen to either be the size of the entire record, a random integer larger than that size, or a random integer smaller than that size.

APE first uses exploration to create a basic model of operation, before modifying the messages. Figure 5 shows APE's model of the TLS heartbeat message behavior. For clarity, the displayed model abstracts the initial handshake as the `-hello` message. APE uses this model and the message modification method to modify the `-heartbeat` message. For each implementation, APE generated 100 traces with message modification and inferred a single model of all the executions.

APE took a total of 713 seconds to analyze the vulnerable server and 251 seconds for the patched server. For both servers, APE needed roughly the same amount of time to generate the 100 traces (106 and 103 seconds) and to search the inferred model for violations (less than 1 second). The time difference lies in the model inference stage — 607 seconds for the vulnerable server and 251 for the patched one. We attribute this result to the greater number of invariants mined from the vulnerable server's traces: 14,952 vs. 7,063.

We omit the final behavioral models APE built for these two servers because of space limitations, but describe our findings. The models contained 158 and 108 states for the vulnerable and patched servers, respectively. When APE's payload modification is enabled, we expect the model size to grow continuously as more traces are added, in contrast to the bounded behavior we observe in Figure 6. This is a consequence of APE's message modification that generates a potentially unbounded number of message types.

We based our violation characteristic directly on language in the protocol specification. Specifically, we set the violation characteristic to be any trace with a heartbeat response that includes more bytes than sum of the `payload` and `padding` of the original message. When exploring the vulnerable server implementation, 22 of the 100 traces included a heartbeat received from the server. Of these 22 paths, 8 included a heartbeat message that triggered the heartbleed vulnerability. For the patched server, 8 of the 100 paths led to heartbeat events, and all of those corresponded to valid heartbeat messages, indicating the patched server was not vulnerable to the heartbleed bug.

APE-assisted, manual specification violation discovery. APE was able to discover the heartbleed vulnerability in the vulnerable server when asked to look for executions that return more data than made available in the original heartbeat message. It also aided the manual discovery of two other specification violations.

We manually examined the final model of the two servers and found two specification violations of which we had no previous knowledge. First, both servers failed to respond to properly formed heartbeat messages with payloads smaller than 4,073 bytes. The protocol specification imposes no such restriction. Second, while the protocol specifies that the server must silently discard the heartbeat message if the total length of the message is greater than 2^{14} bytes, APE observed multiple instances of both servers responding to such messages. We have reported these bugs to the project's developers.

V. EVALUATION: BITTORRENT

This section details the application of APE to three implementations of the BitTorrent protocol. We chose BitTorrent because of its popularity, and because the protocol is implemented by an immeasurable number of clients, allowing us to evaluate APE's effectiveness in testing different implementations of the same protocol. APE found one bug and two unexpected uses of the specification [11] in three popular BitTorrent clients: μTorrent, Azureus, and Transmission. Further, manual exploration of the APE-discovered bug led us to discover two more bugs, one that causes the client to crash, and one that artificially inflates the client's upload rate.

A. The BitTorrent Protocol

BitTorrent is a peer-to-peer file sharing protocol. To share files, a peer creates a small metadata *torrent* file that contains information on finding other peers, file size, and a list of hashes of parts of the file for integrity checking. Peers' BitTorrent clients, after discovering each other via *trackers* that maintain a list of peers' IPs, share files by requesting *pieces* of the file and responding to others' requests. Each piece's integrity can be verified using the torrent's hashes. The piece request messages contain the index, offset, and length of the requested pieces. Pieces are usually downloaded in *chunks* of 16KB, so it may take multiple request messages to download a single piece.

Reciprocation is an important aspect of the BitTorrent protocol. While implementations are free to use whatever algorithm to promote reciprocation, in general, a client is more likely to trade with peers it deems to be contributing. Clients can send a `choke` message to non-contributing peers to stop communication, and may send an `unchoke` message to resume communication. To enable new peers without data to share to download data, BitTorrent clients will periodically send `unchoke` messages to random peers in a process called *optimistic unchoking*. This process allows new peers to send several piece requests and begin downloading pieces even without prior reciprocation.

It is important to note that different BitTorrent clients will behave differently even though they ostensibly implement the same protocol. In part, we can attribute these differences to the developer's implementation decisions. In other words, the BitTorrent protocol specification does not completely define all client behavior; the specification is vague in areas and it deliberately leaves some decisions up to the client developers, e.g., reciprocation strategies. Further, developers may misinterpret even those areas which are clearing defined. These differences are precisely the type of protocol violations APE is designed to detect.

B. Applying APE *to BitTorrent Clients*

We tested three popular BitTorrent clients running on Ubuntu 12.04: μTorrent (v3.0 build 27079), Azureus (4.3.0.6-5), and Transmission (2.51-0-ubuntu-1.3).

To create BitTorrent message definitions for APE to use, we modified an existing open-source implementation, AutonomoTorrent [4], adding logging, additional message types (`bitfield_all`, `bitfield_some`, and `bitfield_none` messages, described below), and hooks for APE to use to send BitTorrent's nine message types [11]. We chose AutonomoTorrent to take advantage of its code for tasks such as getting peer lists from the tracker, checking piece hash values, and file IO. APE users may often choose to reuse existing protocol client implementations to ease the task of writing methods that send protocol messages. Our single message-sending method implementation was sufficient for all three BitTorrent targets, and could be used to test other BitTorrent clients.

During normal operation, peers use the `bitfield` message to advertise their pieces. For APE, we found it useful to split the bitfield into three different message types to indicate if the peer is advertising all, some, or none of the pieces of the file. This distinction is important as BitTorrent peers will display different behavior depending on the bitfield value. For example, by sending a `bitfield_all` message APE is advertising that it already has all of the file pieces; consequently, the other peers will likely never unchoke our APE peer.

During exploration, we configured APE to start the target implementation with half of the file pre-downloaded. This allowed APE to both download and share pieces. We did not use APE's optional message modification capability for BitTorrent.

C. Avoiding Reciprocation

We were interested in searching for specification violations that allow avoiding reciprocation, downloading data without uploading. We thus encoded a violation characteristic to check for the peer downloading pieces without uploading.

APE discovered and verified around 100 execution paths across the three implementations that satisfied this characteristic, meaning it discovered multiple ways of violating this specification. We manually examined all these paths and categorized three distinct strategies for avoiding reciprocation.

1. The choke/unchoke cycle. During normal operation, if the client receives and ignores a `request` message from a peer, that peer sends a `choke` message and no longer responds to messages from the client. However, APE discovered a bug that allows a peer to ignore `request`s without repercussions. Whenever receiving a `request` message, if the client responds by sending a `choke`, followed immediately by an `unchoke` message, (in lieu of the requested piece), the peer sends a new `request` and continues responding to the client's requests. All three tested implementations exhibit this vulnerability.

The next two strategies do not explicitly violate the specification, but result in an unexpected ability to avoid reciprocation.

2. The new guy on the block. The client can pretend to be new to the network by sending a `bitfield_none` messages, even when it actually has data [38]. This falsehood causes other peers to share data without expecting to receive any data in return. Further, APE discovered that to remain a freeloading client, it must never send a `have` message claiming to have the data other peers need.

3. The parrot. An extension of the above strategy is for the client to report in the `bitfield` message only those pieces that the other peers already have.

After APE identified the choke/unchoke cycle, we used APE to apply the exploit of this violation — sending a `choke` and an `unchoke` message in response to each received `request` — to each of the three target BitTorrent implementations. For all three implementations, this exploit caused a drastic increase in both the total number of `request`s (thousands per minute), and the number of distinct `request`s: other peers often requested different pieces. Over a five minute timespan, Azureus, μTorrent, and Transmission sent 172, 593, and 420 distinct piece requests, respectively. Without sending `choke` and `unchoke` messages, those numbers dropped to 4, 5, and 15 distinct piece requests, respectively.

This exploration led us to manually uncover two other related bugs in the μTorrent implementation. First, when running in resource-constrained environments (less than 512MB of RAM), sending many `choke` and `unchoke` messages to a μTorrent client would crash that client. Second, it was possible to manipulate μTorrent clients into thinking a peer's observed upload rate was artificially high because μTorrent included the bandwidth used by the `choke` and `unchoke` messages.

VI. PERFORMANCE EVALUATION

To find vulnerabilities in target implementations, APE must efficiency explore the large state space of the target's possible behavior. Our performance evaluation focuses on the TRACKER server. While TRACKER is small, its size limits the randomness of the search process and allows us to reliably measure how quickly APE finds vulnerabilities. We ran 30 trials of APE discovering vulnerabilities in TRACKER. Each trial consisted of 100 iterations; each iteration used guided exploration to generate one new trace, and then inferred a new Synoptic model of the target's behavior using all observed traces. We use these trials to answer three research questions.

RQ1: How quickly does APE find vulnerabilities in TRACKER?

Recall that the buggy TRACKER server from Section II has two vulnerabilities. First, an attacker can elicit a `+clients_response` by sending a `-clients_request` even without previously sending a `-hello`. Second, an attacker can cause the server to erroneously announce its presence multiple times by sending multiple `-hello` messages. To answer RQ1, we examined each of the 3,000 models generated over the 30 trials to find at which point in the exploration process the model first encoded each of the two vulnerabilities.

The first vulnerability is simple for APE to find since it only needs to generate a model with a path that starts with `-clients_request`. In our experiments, each of the 30 trials found this vulnerability within the first three exploration traces. In 10 of the trials, the first trace revealed the vulnerability. This result is consistent with the random exploration procedure, which dominates APE's guided exploration when the model is

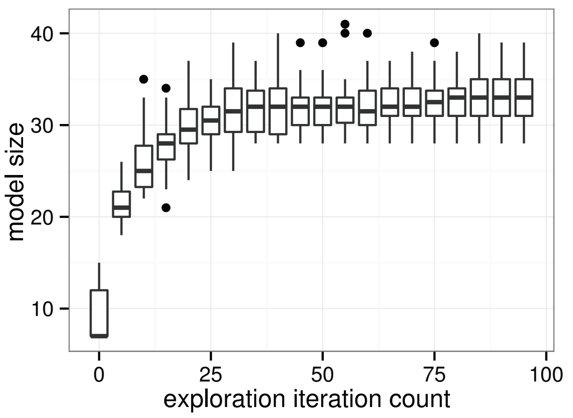

Fig. 6. The size of the model (the sum of the number of states and transitions) approximates the amount of the target's behavior learned by APE. APE's guided exploration learns the behavior and approaches the limit quickly.

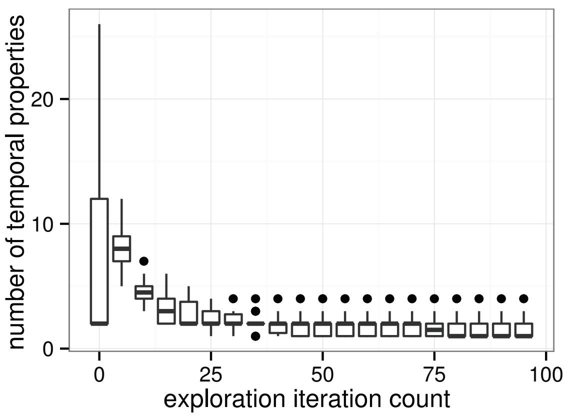

Fig. 7. The accuracy of the model's predictive ability depends on the guided exploration's ability to diversify observed executions and eliminate false temporal invariants. APE accomplishes this and quickly eliminates temporal invariants that are artifacts of small sets of explored executions.

empty or small. The procedure picks, at random, a message to send from the three sendable, previously unsent messages. One third of the time (10/30) APE picks the `-clients_request` messages, and in all instances, after three tries, APE has tried all three messages.

The second vulnerability is harder to find. In our experiments, APE found this vulnerability in each trial after exploring at most six traces. In some cases, the guided exploration generated a trace with multiple `+clients_response` messages. In other cases, model inference *predicted* that such behavior is allowed.

Running APE on TRACKER took no longer than a few minutes on average to discover both vulnerabilities.

RQ2: How quickly does guided exploration learn the target's behavior?

Critical to APE's success is being able to explore its target's behavior quickly. While it likely takes a long time to explore the behavior exhaustively, APE's guided exploration is able to explore a large fraction of the behavior space after relatively few iterations. Using model size (the sum of the number of states and transitions) as an estimate of the measure of how much of the target's behavior has been explored, Figure 6 shows that the mean amount of behavior APE has not learned diminishes exponentially with time. This finding implies that (1) the TRACKER server implementation's behavior measure is bounded, and (2) APE can learn it quickly. Other exploration strategies could learn slowly, forcing more time to be spent in the guided exploration stage, and resulting in poor scaling to larger targets. Note that a threat to validity of this result is that it is possible that some behavior cannot be discovered by APE, so the claim that APE discovers the behavior quickly only applies to that behavior that APE can discover.

RQ3: How does the predictability of the model change as APE explores the target's behavior?

The models APE infers from observed executions are predictive. Model inference can be inaccurate if the temporal invariants that hold for the observed executions do not accurately describe the target system [9]. Thus, for the predictive ability of the model to be accurate, guided exploration must quickly eliminate spurious temporal invariants that are observed only because of lack of diversity of the observed traces.

Figure 7 shows that the mean number of mined temporal invariants diminishes quickly with each iteration, implying that after relatively few iterations, the model accurately predicts behavior.

An astute reader will notice that in the first iterations, the number of temporal invariant increases, whereas it decreases monotonically afterwards. This phenomenon is due to Synoptic's invariant filtering; the inference ignores invariants between messages it has never observed co-occur in a trace. At first, new traces increase the number of messages that have been observed co-occurring, increasing false (and valid) temporal invariants. Further exploration eliminates the false invariants.

While sets of observed executions with poor diversity often satisfy many temporal invariants, systems typically have few true temporal invariants [7], [9]. For TRACKER, the final inferred models satisfied only a single temporal invariant: `-clients_request` is always followed by `+clients_response`.

VII. RELATED WORK

TAUTOKO [22], Xie and Notkin [49], ProCrawl [43], and MACE [16] are the most similar approaches to APE. TAUTOKO explores the execution space and infers behavioral models, but is limited to data structure models and method call events. Xie and Notkin combines model inference and execution to generate new tests, but focuses on single classes and unit tests [49]. ProCrawl infers behavioral models of web applications by guiding the exploration using model inference. By contrast, APE targets networked systems and allows modifying messages. MACE detects vulnerabilities using symbolic and concrete execution, guided by model inference, which is complementary to APE's dynamic approach. Other related prior work [18], [27], [30] had relied on user-specified descriptions of an input abstraction function, whereas MACE does not, but still requires an output abstraction function for the output. These are similar to APE's message definitions.

APE uses Synoptic [9], which relies on mining temporal invariants to abstract the observed executions and balance running time against the conciseness of the final model.

Synoptic starts with an overgeneralized, compact model and iteratively refines the model to eliminate violations of the observed temporal invariants. Synoptic is nondeterministic, which adds to the nondeterminism already inherent in APE, although deterministic model inference [6], [7] could be used instead.

Model-inference research is complementary to our work and APE can directly benefit from improvements in model inference. APE uses predictive model inference to abstract the observed executions into a concise and predictive model. In general, inferring the most concise model from observed examples is NP-complete [1], [17], [26], and most model-inference algorithms approximate a solution [6], [7], [8], [9], [10], [15], [20], [22], [23], [25], [31], [32], [34], [37], [39], [42], [47], [49].

The kTails model inference algorithm [10] is the basis for numerous other model-inference algorithms [15], [20], [31], [35], [36], [37], [39], [42], [47]. Unlike Synoptic, kTails starts with the set of observed traces, represents these traces as an FSM made up of a set of linear graphs, and then iteratively coarsens the FSM by merging states that are identical in the k subsequent states.

Synoptic relies on temporal invariants inferred from the observed traces. The accuracy of this mining and the richness of the invariants directly affects the model accuracy and APE's ability to precisely discover specification violations. There are numerous algorithms that mine temporal property instances [2]. For example, Javert [24] is a temporal specification mining tool that infers specifications by composing simpler micropatterns into larger ones. Javert's focus is on implementing this composition efficiently. Synoptic could use Javert to improve efficiency or richness of its mined invariants.

Some model inference algorithms infer models that are extensions to FSMs. GK-Tails [39] requires EFSMs and RPNI [15] requires Probabilistic FSMs. APE currently uses FSMs but can be extended to use these extended models and model inference algorithms. Other model inference algorithms may require other extensions that APE can likely be extended to handle.

APE's payload modification is similar to protocol fuzz testing [5], [27], [30], but APE guides exploration, instead of randomly searching through the state space. APE also aims to automate more of the exploration than, for example, SNOOZE [5] and Gorbunov et al. [27].

Protocol reverse engineering [13], [14], [21] automates inferring message format, which can be used to automate APE even further. Prospex [18] combines reverse engineering message format and fuzz testing and can generate test inputs, but unlike APE, does not automatically discover specification violations.

Gatling finds performance attacks in large-scale distributed networks [33]. Like APE, Gatling simulates behavior on the network with peers, although using multiple peers is APE's future feature at this time. While Gatling allows changing message fields and dropping messages, APE is more general in its message modifications. Meanwhile some systems focus on proving specific protocol properties, such as authentication and authorization properties [29]. By contrast, APE is more general and can discover any specification violation that can be described by a violation characteristic method.

VIII. CONTRIBUTIONS

We have presented APE, a technique for discovering and verifying specification violations in networked software. APE explores the large space of behavior by dynamically inferring precise models of behavior, stimulating unobserved behavior likely to lead to violations, and iterating by refining the behavior models with the new, stimulated behavior.

We publicly released an open-source APE prototype: `http://forensics.umass.edu/ape.php`.

In evaluating APE, we verified the known heartbleed vulnerability in OpenSSL, and found seven other specification violations or unexpected behaviors in OpenSSL and three popular BitTorrent clients. APE can both discover violations and help developers better understand system behavior. Our prototype implementation and its evaluation show great promise for using model inference, together with fuzz testing, to find bugs, verify bug patches, and identify related exploits of known bugs, and augment and generate test suites.

IX. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under grants CCF-1453508, CNS-1513055, and CNS-1018615.

REFERENCES

[1] D. Angluin. Finding Patterns Common to a Set of Strings. *Journal of Computer and System Sciences*, 21(1):46–62, 1980.

[2] C. M. Antunes and A. L. Oliveira. Temporal data mining: An overview. In *Proceedings of the Workshop on Temporal Data Mining*, 2001.

[3] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.

[4] AutonomoTorrent. http://github.com/abhijeeth/AutonomoTorrent.

[5] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. SNOOZE: Toward a stateful network protocol fuzzer. *Information Security*, pages 343–358, 2006.

[6] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying fsm-inference algorithms through declarative specification. In *International Conference on Software Engineering (ICSE)*, pages 252–261, May 2013.

[7] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Using declarative specification to improve the understanding, extensibility, and comparison of model-inference algorithms. *IEEE Trans. on Software Engineering (TSE)*, 41(4):408–428, April 2015.

[8] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *International Conference on Software Engineering (ICSE)*, pages 468–479, June 2014.

[9] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 267–277, September 2011.

[10] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Transactions on Computers (TC)*, 21(6):592–597, 1972.

[11] The BitTorrent Protocol Specification. http://www.bittorrent.org/beps/bep_0003.html, 2012.

[12] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[13] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *ACM Conference on Computer and Communications Security*, pages 621–634, 2009.

[14] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *ACM conference on Computer and Communications Security*, pages 317–329, 2007.

[15] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference and Applications*, pages 139–152, 1994.

[16] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security*, volume 11, 2011.

[17] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[18] P. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *IEEE Security and Privacy*, pages 110–125, 2009.

[19] Consumer Reports. Online exposure. Social networks, mobile phones, and scams can threaten your security. *Consumer Reports Magazine*, June 2011.

[20] J. E. Cook and A. L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.

[21] W. Cui, J. Kannan, and H. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.

[22] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering (TSE)*, 38(2):243–257, 2012.

[23] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *International Workshop on Dynamic Analysis (WODA)*, pages 17–24, 2006.

[24] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[25] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining behavior models from user-intensive web applications. In *International Conference on Software Engineering (ICSE)*, pages 277–287, 2014.

[26] E. M. Gold. Language Identification in the Limit. *Information and Control*, 10(5):447–474, 1967.

[27] S. Gorbunov and A. Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *International Journal of Computer Science and Network Security IJCSNS*, 10(8):239, 2010.

[28] CVE-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, 2014.

[29] Y. Hsu and D. Lee. Authentication and authorization protocol security property analysis with trace inclusion transformation and online minimization. In *International Conference on Network Protocols (ICNP)*, pages 164–173, 2010.

[30] Y. Hsu, G. Shu, and D. Lee. A model-based approach to security flaw detection of network protocol implementations. In *International Conference on Network Protocols*, pages 114–123, 2008.

[31] I. Krka, Y. Brun, and N. Medvidovic. Automatic mining of specifications from invocation traces and method invariants. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 178–189, 2014.

[32] T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh. Synergizing specification miners through model fissions and fusions. In *International Conference on Automated Software Engineering (ASE)*, 2015.

[33] H. Lee, J. Seibert, C. Killian, and C. Nita-Rotaru. Gatling: Automatic attack discovery in large-scale distributed systems. In *USENIX Network & Distributed System Security Symposium*, 2012.

[34] W. Li, L. Dworkin, and S. A. Seshia. Mining assumptions for synthesis. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 43–50, 2011.

[35] D. Lo and S.-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Working Conference on Reverse Engineering (WCRE)*, 2006.

[36] D. Lo and S.-C. Khoo. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 265–275, 2006.

[37] D. Lo, L. Mariani, and M. Pezzè. Automatic Steering of Behavioral Model Inference. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 345–354, 2009.

[38] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *HotNets*, 2006.

[39] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic Generation of Software Behavioral Models. In *International Conference on Software Engineering (ICSE)*, pages 501–510, 2008.

[40] T. Ohmann, M. Herzberg, S. Fiss, A. Halbert, M. Palyart, I. Beschastnikh, and Y. Brun. Behavioral resource-aware model inference. In *International Conference on Automated Software Engineering (ASE)*, pages 19–30, September 2014.

[41] T. Ohmann, K. Thai, I. Beschastnikh, and Y. Brun. Mining precise performance-aware behavioral models from existing instrumentation. In *New Ideas and Emerging Results Track at the International Conference on Software Engineering (ICSE)*, pages 484–487, June 2014.

[42] S. P. Reiss and M. Renieris. Encoding Program Executions. In *International Conference on Software Engineering (ICSE)*, pages 221–230, 2001.

[43] M. Schur, A. Roth, and A. Zeller. Mining behavior models from enterprise web applications. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 422–432, 2013.

[44] Symantec. Symantec Internet security threat report. Trends for January 06–June 06. http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf, September 2006.

[45] Symantec. Symantec Internet security threat report. Trends for 2010. https://www4.symantec.com/mktginfo/downloads/21182883_GA_REPORT_ISTR_Main-Report_04-11_HI-RES.pdf, April 2011.

[46] Python — Twisted. http://twistedmatrix.com/.

[47] N. Walkinshaw and K. Bogdanov. Inferring finite-state models with temporal constraints. In *International Conference on Automated Software Engineering (ASE)*, pages 248–257, 2008.

[48] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228, 2002.

[49] T. Xie and D. Notkin. Mutually enhancing test generation and specification inference. In *International Workshop Formal Approaches to Testing of Software (FATES)*, pages 60–69, 2003.