Compositional Testing of Internet Protocols

Kenneth L. McMillan
Microsoft Research, Redmond
Email: kenmcmil@microsoft.com

Lenore D. Zuck
Department of Computer Science, UIC
Email: lenore@cs.uic.edu

***Abstract*—We introduce a methodology of Network-centric Compositional Testing (NCT) to develop formal wire specifications of Internet protocols and to test protocol implementations for compliance to a common standard. We use formal specifications to generate automated testers for implementations of the protocol, based on randomized constraint solving using an SMT solver. This makes it possible to resolve ambiguities in informal standards documents using knowledge inherent in the implementations, while at the same time testing the implementations for compliance to the developing formal specification. Because the testing is compositional, it allows us to detect cases when the specification is either too weak or too strong, and to refine the specification accordingly. We apply the methodology to QUIC, a new Internet secure transport protocol currently in the process of IETF standardization and intended as a replacement for the TLS/TCP stack and a foundation for HTTP/3. In the process of specifying QUIC, we discovered numerous errors in implementations, as well as issues in the standard itself. These include an off-path denial of service attack and an information leak similar to the "heartbleed" vulnerability in OpenSSL. The paper describes the formal foundations of the methodology, and summarizes its specific application to QUIC.**

***Index Terms*—Specification based conformance testing, Network Protocols, Light-weight Formal Methods, QUIC**

I. Introduction

Internet protocols are developed in the form of RFCs: English-language documentation that provides extensive guidance for implementers of the protocol, but is nonetheless ambiguous and broadly open to interpretation. The primary mechanism for resolving these ambiguities and validating the correctness of the protocol design is to produce multiple independent implementations, and to test these implementations for interoperability. As a result, these implementations represent a kind of commentary on the standard document, providing concrete interpretations where the language may be vague, unclear or contradictory.

While effective, this methodology leaves something to be desired from the point of view both of clear standardization and of implementation compliance. First, the knowledge implicit in the implementations is not captured in any precise and rigorous way. Second, since the implementations do not represent the *full* diversity of behaviors that the protocol allows, interoperability testing provides very limited test coverage of protocol behaviors. Third, because actual protocol compliance is never tested, interoperability is *not* sufficient to guarantee that current implementations will be interoperable with future implementations meeting the standard. The result is that implementations in the wild become the *de facto* standard. This effect has important security consequences, as illustrated by the history of SSL/TLS. Non-compliant implementations of this protocol in the wild led to numerous vulnerabilities, due for example to vulnerable work-arounds in clients [1]–[3].

In this paper, we argue that it is important to develop an unambiguous formal statement of a protocol standard, in a form that allows implementations to be effectively tested for actual *compliance to the standard*, and not just for interoperability. Moreover, it is necessary to test implementations in *adversarial environments* and not just in the benign environment of other existing implementations.

We introduce a methodology we call Network-centric Compositional Testing (NCT) to serve this purpose. In NCT, formal specifications are used to generate automated testers for implementations of the protocol, based on randomized constraint solving using an SMT solver. Randomized generation is highly effective in producing *adverse stimulus*, that is, behaviors in which messages occur outside of the expected order, or with unusual combinations of parameter values. This makes it possible to resolve ambiguities in informal standards documents using knowledge inherent in the implementations, while at the same time testing the implementations adversarially for compliance to the developing formal specification.

By 'network-centric', we mean that the specification describes the protocol in terms of its behaviors *as observed on the wire* and not as an abstract implementation of the protocol. This allows us to test the protocol *compositionally*. That is, any *assumptions* made on the input of one process in the protocol are treated as *guarantees* on the output of other process. Without compositionality, we cannot infer from the fact that implementations pass all tests that they will interoperate correctly when composed. Moreover, while monitoring protocol traces using a formal specification allows us to discover cases where the specification is too strong (*i.e.*, it rejects a legal protocol behavior) compositional testing can reveal cases where the specification is too *weak*. That is, suppose the specification of an output of a protocol node is too weak. This same specification is used to generate inputs for its peer. Thus, we can detect the weakness by the fact that the peer misbehaves or flags a protocol error on a generated input.

NCT addresses two primary challenges in network protocol specification. First, its compositional rule (Sec. II) is applicable to testing protocols on the Internet. Notably, it supports

The work of the L. D. Zuck was partially supported by NSF awards CCF-1564296 and CCF-1918429

978-1-5386-7289-1/19/$31.00 ©2019 IEEE
DOI 10.1109/SecDev.2019.00031

protocols with an arbitrary number of participants, in an environment in which there are no pre-established channels or process identities. Second, we introduce optimized algorithms for randomized constraint solving, and corresponding specification strategies, that are suitable for generating test traffic with complex message structures and large data transfers (Sec. III,IV). The methodology can thus be applied to protocol implementations that are not cleanly layered, and require us to test the complete protocol stack, with its attendant deeply nested encapsulation of messages.

To evaluate NCT, we applied it (Sec. VI) to QUIC, a new Internet secure transport protocol introduced by Google as a replacement for the TLS/TCP stack, and currently in the process of IETF standardization. Because QUIC has been selected as the foundation for HTTP/3, the next official version of the hypertext transfer protocol, it is reasonable to expect that the protocol will soon carry a significant portion of Internet traffic. In the process of developing a network-centric specification for QUIC, the adverse stimulus produced by randomized testing revealed numerous errors in the current implementations, as well as issues in the standard itself, that were not discovered by directed tests or interoperability testing. These include an off-path denial of service attack and an information leak similar to the "heartbleed" vulnerability in OpenSSL. By exposing previously unseen implementation behaviors, we discovered vulnerabilities, even though we did not explicitly search for them. For this reason, we think the methodology may also be valuable to security teams explicitly searching for vulnerabilities.

Related work. There are numerous approaches to verification or adversarial testing of network protocol implementations. Many of these do not check compliance to a common formal protocol standard (*e.g.*, [4]–[7]).[1] This includes some interesting recent work applying white-box testing to QUIC [8]. Techniques that *do* address formalization and compliance include model-based testing (MBT) [9]–[11] and its precursors in the area of protocol conformance testing [12]. These methods assume that specifications are finite-state machines (FSMs) or can be effectively restricted to be finite-state. They use systematic exploration of the FSM as a heuristic for adversarial test generation. NCT differs from these methods in two primary respects. First, it is compositional, with the benefits we noted above. Second, it does *not* assume the specification is an FSM. This is crucial for protocols such as QUIC that are inherently not finite-state, and allows us to capture all aspects of the protocol needed for full interoperation with real implementations. This is in contrast to, *e.g.*, [2], [10], [11], which capture only certain finite-state aspects of a protocol.

Another effort that infers protocol specifications experimentally from implementations is the Network Semantics Project [13] which has developed a formal specification of TCP. In this work, the formal specification is used to monitor traces captured on the wire. It is not used for test generation and is not compositional.

[1]Note that [6] formally proves security properties of a reference implementation, but does not prove compliance to a common standard.

II. COMPOSITIONAL TESTING FOR INTERNET PROTOCOLS

In compositional reasoning, we have a system that consists of a collection of processes communicating in some way. Each process has a *local specification* that determines its allowed input/output behavior. This is also called the *guarantee* of the process. In addition, each process is allowed to make *assumptions* about its environment (that is, the other processes in the system). As observed originally by Chandy and Misra [14], because the processes are interconnected, the assumptions of one process are the guarantees of the other processes. To avoid making a circular argument, we have to define carefully the assumptions that a process may make about its environment. For example, we say that the correctness of a given output produced by a process may depend on the the correctness of inputs received in the past, but not those that will be received in the future.

A typical rule for compositional reasoning (as used in [15]) looks like this:

$$\frac{\langle \phi_2 \rangle\ \pi_1\ \langle \phi_1 \rangle \qquad \langle \phi_1 \rangle\ \pi_2\ \langle \phi_2 \rangle}{\langle \mathsf{true} \rangle\ \pi_1 \parallel \pi_2\ \langle \phi_1 \wedge \phi_2 \rangle} \tag{1}$$

Here the triple $\langle \phi \rangle \pi \langle \psi \rangle$ means that, if the environment of process π satisfied the assumption ϕ always in the past, process π guarantees property ψ at the present moment. The premises of the rule state that each process maintains its guarantee so long as the other does (in fact, one step longer). From these premises, we can conclude that neither guarantee is ever violated when we compose the two processes together, since neither guarantee can be the first to be false.

In a compositional testing approach [16], [17], rather than formally verify the premises of Rule 1, we simply test the actual artifacts π_1 and π_2. For each process, we *generate* inputs satisfying its assumption, and we *check* that the resulting behavior satisfies the guarantee. Rule 1 provides a formal proof, but the premises of this proof are tested empirically.

For testing Internet protocols, however, the rule above is significantly inconvenient. The main reason is that Internet protocols involve an arbitrary number of processes distributed across a network. For example, a server may connect to an arbitrary number of clients, or a peer-to-peer protocol may connect an arbitrary number of peers. When we generalize Rule 1 to an unbounded collection of processes, we find that each process now makes an unbounded number of assumptions (that is, it assumes that all other processes in the network follow their local specifications). For purposes of formal proof, this may not be problematic. However, if we wish to generate test inputs that satisfy this infinite collection of properties, we have a problem.

Our approach to this problem is to replace the infinite collection of local specifications with a single global specification ϕ that fully describes the protocol. That is, ϕ determines, for any global trace of protocol messages, whether the protocol has

been followed. Of course, no one local process can guarantee this global property. Rather, we define what it means for a single process π_i to *cause* the failure of the global property. This occurs when a single output produced by π_i changes the status of the property from true to false. We interpret the triple $\langle\phi\rangle\ \pi_i\ \langle\phi\rangle$ to mean that π_i does not *cause* the specification ϕ to be falsified. If all the local processes have this property, then the specification must hold. Our local testing problem is now to generate inputs for π_i that do not cause ϕ to fail, and to check that the resulting outputs of π_i do not cause ϕ to fail. In the sequel, we will formalize these notions.

A more subtle reason to use a global specification is that a global specification can be monitored on the network. Local specifications, as in Rule 1, assume that the communications of each process can be identified. On the Internet, however, we cannot reliably identify the source of messages. In fact, the purpose of many protocols is precisely to establish the identity of communicating parties. In particular, we cannot assume a one-to-one correspondence between processes and network addresses since a process may use many addresses, and these may change over time (*i.e.*, the process may be *mobile*). Moreover, the same address may be used by many processes at different times, or even at the same time in the case of malicious processes that 'spoof' addresses. A network address thus provides a heuristic for 'best effort' delivery of a message to an intended recipient, but the network provides no guarantees regarding the parties that ultimately observe a message. For this reason, it is useful to write a protocol specification as a global property that does not refer to the source of messages, but only describes the messages that may be sent by *any* process at a given time. We will refer to such specifications as *process-oblivious*.

There are two other important testing-related issues that determine the form of our compositional reasoning system. For purposes of assume/guarantee testing, we must effectively be able to use our specifications as both *generators* of inputs and *checkers* of outputs of processes. This motivates the use of a deterministic guarded command formalism for property specification instead of, for example, a temporal logic or non-deterministic labeled transition systems. Moreover, rather than decompose the system into a collection of disjoint processes, we decompose it into a collection of overlapping subsets of processes called *locales*. As we will see, this allows us to generate tests in cases where some system components lack formal specifications but do have concrete implementations. All of these aspects distinguish our system from prior compositional systems such as [14], [16], [18], [19].

Example 1. As a running example, we will use a toy protocol among an arbitrary number of clients and servers. A client, say Alice, sends a request message to a server, say Bob, that contains a fresh nonce value N that acts as a connection identifier, and a data value D. The server simply echoes this information in a response message. Our specification requires that request messages be consistent, in the sense that no two requests with the same connection identifier have different data values, and responses match previous requests.

A. Processes

In our model, processes communicate values *via* channels. Each channel is an output of exactly one process, but it may be an input of many processes. Communication over channels is synchronous, so that a value is received at the same instant in which it is transmitted. To model asynchronous communication over a network, we will use an explicit process to model the network.

Let $\mathcal{C}$ be a (possibly infinite) set of *channels*. We model the content of a message on a channel $c \in \mathcal{C}$ by a valuation of a set of parameters V_c. Each parameter $v \in V_c$ has an associated range R_v. A *value* of channel c maps every parameter $v \in V_c$ to an element of R_v. We denote the set of such mappings by R_c.

An *event* is a pair (c, V), where $c \in \mathcal{C}$ and $V \in R_c$. An event corresponds to transmission of a value on a channel. A *trace* over a set of channels $C \subseteq \mathcal{C}$ is a finite sequence of events $(c_1, V_1), \ldots, (c_k, V_k)$ where $c_i \in C$ for $1 \leq i \leq k$. A trace represents an I/O behavior of a process. We write *Traces*(C) for the set of traces over C. We will call a trace over the full set of channels $\mathcal{C}$ simply a *trace*.

A *process* is a triple (I, O, T), where $I \subseteq \mathcal{C}$ is the input set, $O \subseteq \mathcal{C}$ is the output set and T is a prefix-closed set of $(I \cup O)$-traces. (A set of sequences is *prefix closed* if for every sequence in the set, all its prefixes are also in the set.)

Example 2. Each process π in our model of the toy protocol has a single input channel $\text{R}_{\text{CV}\pi}$ and a single output channel $\text{S}_{\text{ND}\pi}$. All channels have three parameters: a message type M ranging over the set $\{\text{R}_{\text{EQ}}, \text{R}_{\text{SP}}\}$, a connection identifier N and a data value D (where N and D are, say, natural numbers). As a shorthand, we write a send event as $\text{S}_{\text{ND}\pi}(M, N, D)$ and a receive event as $\text{R}_{\text{CV}\pi}(M, N, D)$. Some example traces of a client process A are:

$\text{S}_{\text{ND}A}(\text{R}_{\text{EQ}}, 42, 0), \text{R}_{\text{CV}A}(\text{R}_{\text{SP}}, 42, 0), \text{S}_{\text{ND}A}(\text{R}_{\text{EQ}}, 43, 1), \text{R}_{\text{CV}A}(\text{R}_{\text{SP}}, 43, 1)$
$\text{R}_{\text{CV}A}(\text{R}_{\text{EQ}}, 42, 0), \text{R}_{\text{CV}A}(\text{R}_{\text{SP}}, 43, 1), \text{S}_{\text{ND}A}(\text{R}_{\text{EQ}}, 44, 2)$

The first trace shows the normal client sequence, in which a request is followed by a corresponding response, then another request/response cycle begins. The second trace shows that the client may receive unexpected messages, which it ignores. Some example traces of a server B are:

$\text{R}_{\text{CV}B}(\text{R}_{\text{EQ}}, 42, 0), \text{R}_{\text{CV}B}(\text{R}_{\text{EQ}}, 43, 1), \text{S}_{\text{ND}B}(\text{R}_{\text{SP}}, 43, 1), \text{S}_{\text{ND}B}(\text{R}_{\text{SP}}, 42, 0)$
$\text{R}_{\text{CV}A}(\text{R}_{\text{EQ}}, 42, 0), \text{R}_{\text{CV}B}(\text{R}_{\text{SP}}, 42, 1), \text{S}_{\text{ND}B}(\text{R}_{\text{EQ}}, 42, 1)$

Again, the first is a normal sequence. In the second, the server receives two conflicting requests. Finally, consider a process *Net* representing the network. The inputs of *Net* are the send channels $\text{S}_{\text{ND}\pi}$, while the outputs are the receive channels $\text{R}_{\text{CV}\pi}$. Some example traces of *Net* are:

$\text{S}_{\text{ND}B}(\text{R}_{\text{EQ}}, 42, 0), \text{R}_{\text{CV}B}(\text{R}_{\text{EQ}}, 42, 0), \text{R}_{\text{CV}A}(\text{R}_{\text{SP}}, 43, 1)$
$\text{S}_{\text{ND}B}(\text{R}_{\text{EQ}}, 42, 0), \text{S}_{\text{ND}B}(\text{R}_{\text{EQ}}, 42, 0), \text{S}_{\text{ND}B}(\text{R}_{\text{EQ}}, 42, 0)$

The network provides no guarantees about delivery, except that it will not invent messages. □

The *projection* of a trace t onto a set of channels $C \subseteq \mathcal{C}$, denoted $t \downarrow C$ is the subsequence of t consisting of all events (c, V) such that $c \in C$.

Suppose that, for $i = 1, 2$, $\pi_i = (I_i, O_i, T_i)$. The processes π_1 and π_2 are *composable* if $O_1 \cap O_2 = \emptyset$. If the two processes are composable, then their *composition*, written $\pi_1 \parallel \pi_2$, is defined by the process (I, O, T) where:

- $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$
- $O = (O_1 \cup O_2)$
- $T = \{t \in \mathit{Traces}(I \cup O) \mid t \downarrow (I_i \cup O_i) \in T_i \text{ for } i = 1, 2\}$

Example 3. Consider the composition $A \parallel \mathit{Net} \parallel B$, where A is a client and B is a server. The input set of this composition is empty, while its output set is $\{\textsc{Snd}_A, \textsc{Rcv}_A, \textsc{Snd}_B, \textsc{Rcv}_B\}$. An example of a trace of the composition is:

$$\textsc{Snd}_A(\textsc{Req}, 42, 0), \textsc{Rcv}_B(\textsc{Req}, 42, 0), \textsc{Snd}_B(\textsc{Rsp}, 42, 0), \textsc{Rcv}_B(\textsc{Rsp}, 42, 0)$$

Notice that the projection of this trace onto the alphabet of each of the three processes is trace of that process. For example, the projection onto A is:

$$\textsc{Snd}_A(\textsc{Req}, 42, 0), \textsc{Rcv}_A(\textsc{Req}, 42, 0)$$

which is a trace client, while the projection onto B is:

$$\textsc{Rcv}_B(\textsc{Req}, 42, 0), \textsc{Snd}_B(\textsc{Rsp}, 42, 0)$$

which is a server trace. Moreover, the entire trace is a trace of *Net*, since every received message was previously sent. □

The $\parallel$ operator is commutative/associative. We generalize the notion of composition to sets of processes. Let Π be an indexed family of processes $\pi_i = (I_i, O_i, T_i)$, where i ranges over a (possibly infinite) set, such that the processes in Π are pair-wise composable. The parallel composition $\parallel \Pi$ is the process (I, O, T) where $I = (\cup_i I_i) \setminus (\cup_i O_i)$, $O = \cup_i O_i$, and T is the set of traces t in $\mathit{Traces}(I \cup O)$ such that for every i, the projection of t onto $I_i \cup O_i$ is a T_i-trace.

B. Specifications

We express a safety property in the form of a machine that accepts a language of finite traces of events. This machine is represented using a parameterized form of guarded commands called *actions*. An action has a *guard* that determines whether a given event is allowed in a given state, and an *update* that modifies the state deterministically as a function of the event. A set of actions, combined with an initial state, accepts a prefix-closed language of finite sequences of events.

The state of a safety specification is represented by a collection of function and relation symbols. A state is thus a first-order structure, giving a valuation of these symbols. The values of the function and relation symbols capture information about the history of events. The guard is a first-order logic formula over the event parameters and the state symbols. It determines when an event is enabled in a given state. This will allow us to generate events that satisfy the guards using an SMT solver.

Let Σ be a *signature* of sorted first-order symbols. A *state* is a Σ-structure, that is, a valuation of the state symbols. We will write $\mathit{structs}(\Sigma)$ for the set of all Σ-structures. We also assume a special constant symbol $\hat{p} \notin \Sigma$ of a sort that ranges over channels.

A Σ-*action* is a tuple $\alpha = (\Lambda, \Sigma_a, \gamma, \tau)$, where:

1) $\Lambda \subseteq \mathcal{C}$ is a set of channels,
2) the *parameter signature* Σ_a such that $\Sigma_a = V_c$ for all $c \in \Lambda$,
3) the *guard* γ is a first-order formula over $\Sigma \cup \Sigma_a \cup \{\hat{p}\}$,
4) the *update* τ is a function from $\mathit{structs}(\Sigma \cup \Sigma_a \cup \{\hat{p}\})$ to $\mathit{structs}(\Sigma)$.

Notice that (2) implies that all of the channels associated to an action must have the same parameter set. Given a Σ-action $\alpha = (\Lambda, \Sigma_a, \gamma, \tau)$, an α-*event* is an event (c, V) where $c \in \Lambda$ and V is a Σ_a-structure. An α-event $e = (c, V)$ is *enabled* in a state s if $\langle s, V, \hat{p} \mapsto c\rangle \models \gamma$. Here we use $\langle s, V\rangle$ to denote the structure s augmented by an assignment V.

Given an α-action as above, an α-*transition* is a triple (s, e, s'), where s and s' are states, $e = (c, V)$ is an α-event enabled in s, and $\tau\langle s, V, \hat{p} \mapsto c\rangle = s'$.

A *specification* is a triple $(\Sigma, s_0, \mathcal{A})$ where Σ is a first-order state signature, the initial state s_0 is a Σ-structure and $\mathcal{A}$ is a set of Σ-actions such that the channel sets of distinct actions are disjoint.

Example 4. In the specification of our toy protocol, the state signature Σ consists of a ternary relation S that records the set of messages (M, N, D) that have been sent on the network. In the initial state, the relation is empty. We will write an action $(\Lambda, \Sigma_a, \gamma, \tau)$ in the form $\Lambda(\Sigma_a) : \gamma \to \tau$, where τ is expressed as a sequence of assignments that update the state symbols. The actions of our toy specification are:

$$\begin{array}{lll} \textsc{Snd}(M, N, D): & (\forall x.(S(M, N, x) \Rightarrow x = D)) & \\ & \quad \wedge (M = \textsc{Rsp} \Rightarrow S(\textsc{Req}, N, D)) & \\ & \quad \to & \{S(M, N, D) := \mathsf{true}\} \\ \textsc{Rcv}(M, N, D): & S(M, N, D) & \to \quad \{\} \end{array}$$

The guard for send events says two things about sent messages. First, the values must be *data-consistent*, that is, if a sent message shares a type and connection id with previous message, it must also have the same data. The second is that a response message must match a previously sent request. Upon a send event, the relation S is updated to indicate that the message was sent. We require of the network that receive events must match prior send events, but nothing else (*i.e.*, messages may be duplicated or reordered). □

Let ϕ be a safety specification $(\Sigma, s_0, \mathcal{A})$. A *transition* of ϕ is an α-transition of any action $\alpha \in \mathcal{A}$. A *run* of ϕ is a finite sequence of transitions $t_0, t_1, \ldots$ where each $t_i = (s_i, e_i, s_{i+1})$ and s_0 is the initial state. A *trace* of ϕ is the sequence of events $e_0, e_1, \ldots$ corresponding to some run $t_0, t_1, \ldots$ If t is a trace of ϕ, we write $t \models \phi$. If $\pi = (I, O, T)$ is a process, we write $\pi \models \phi$ to indicate that for all $t \in T$, $t \models \phi$.

Note that, since the update function of a specification is deterministic, we can easily check whether $t \models \phi$. That is, a trace defines a unique run, and we have only to verify that the guards along this run are satisfied. Without determinism, we

might have to consider an infinite collection of possible runs to determine that $t \not\models \phi$. Because we need to use ϕ operationally as a checker, we require that it be deterministic. This does not mean, however, that the *processes* in our system must be input-output deterministic.

Using deterministic acceptors as specifications has other advantages. Given two safety specifications ϕ and ϕ', we can use standard automata-theoretic constructions to build safety specifications corresponding to $\phi \wedge \phi'$ and $\phi \Rightarrow \phi'$.

C. Compositional Proofs

We now introduce our notion of *assume/guarantee* specification. Intuitively, we intend the specification $\langle\psi\rangle\ \pi\ \langle\phi\rangle$ to mean that, if process π violates property ϕ, then property ψ must have been violated in the past. That is, a π must maintain the guarantee ϕ so long as the environment maintains the assumption ψ. However, because our specifications are global, applying to all events in the system, we must be careful about what we mean by "π violates ϕ". We intend this to mean that the *cause* of the failure of ϕ is an *output* of π.

Let $\pi = (I, O, T)$ be a process. We say that trace t is π-*terminal* if t ends in an event (c, V) such that $c \in O$.

Definition 1: Let ϕ and ψ be specifications and $\pi = (I, O, T)$ a process. We say that $\langle\phi\rangle\ \pi\ \langle\psi\rangle$ holds when, for every π-terminal trace t such that $t \downarrow (I \cup O) \in T$ and $t \not\models \psi$, there exists a strict prefix t' of t such that $t' \not\models \phi \wedge \psi$.

Notice that the triple $\langle\phi\rangle\ \pi\ \langle\phi\rangle$ means essentially that π is not the cause of the failure of ϕ. That is, π is not the first process whose output causes ϕ to be false. We say a process is *closed* if its input set is empty. In a closed system, if no process is the cause of the failure of ϕ, then ϕ always holds. We can express this idea with the following inference rule:

Theorem 1: The following inference rule is sound, provided $\pi_1 \parallel \pi_2$ is closed:

$$\frac{\langle\phi\rangle\ \pi_1\ \langle\phi\rangle \quad \langle\phi\rangle\ \pi_2\ \langle\phi\rangle}{\pi_1 \parallel \pi_2 \models \phi} \tag{2}$$

Proof. Assume a trace t of $\pi_1 \parallel \pi_2$ that violates ϕ, and is a shortest such trace. Since $\pi_1 \parallel \pi_2$ is closed, t must be either π_1- or π_2-terminal. Without loss of generality, assume it is π_1-terminal. By assumption, every strict prefix of t satisfies ϕ. This contradicts $\langle\phi\rangle\ \pi_1\ \langle\phi\rangle$. □

This idea of assigning the cause of a property failure to a process can be extended to subsets of processes. A *localization* L of a set of processes Π is a set of subsets $L_i \subseteq \Pi$ such that $\cup_i L_i = \Pi$ (that is, it is a union-decomposition of Π). The sets L_i are called *locales*. In a closed system, every event is the output of at least one locale. Thus, if no locale causes a property ϕ to fail, then ϕ must always hold. We can formalize this idea with the following proof rule:

Theorem 2: Given a composable set of processes Π, such that $\parallel \Pi$ is closed, and a localization L of Π, the following inference rule is sound:

$$\frac{\text{for all } \ell \in L\text{:}\quad \langle\phi\rangle\ \parallel \ell\ \langle\phi\rangle}{\parallel \Pi \models \phi} \tag{3}$$

Proof Let t be trace of $\parallel \Pi$ that violates ϕ and is a shortest such trace. Every strict prefix of t satisfies ϕ. Since every process is in at least one locale, it follows that there is some locale $\ell \in L$ such that t is $\parallel \ell$-terminal. This contradicts $\langle\phi\rangle\ \parallel \ell\ \langle\phi\rangle$. □

Example 5. Locales are useful in compositional testing if we have some component of a system for which a strong enough specification cannot be obtained. This is a very common situation, for example, when using third-party libraries. As an example, suppose we have a system of three processes A, B, C with corresponding output channels O_A, O_B, O_C. In our specification ϕ, the guard for O_C events is too weak (perhaps it is just `true`) to act as an assumption for processes A and B. We can still verify this system compositionally by reasoning as follows:

$$\frac{\langle\phi\rangle\ A \parallel C\ \langle\phi\rangle \quad \langle\phi\rangle\ B \parallel C\ \langle\phi\rangle}{A \parallel B \parallel C \models \phi}$$

That is, we include the unspecified process C in both locales. This allows A and B to rely on unspecified properties of C. In this proof, we say C is "self-specified". □

D. Mirrors

In order to test whether the triple $\langle\phi\rangle\ \pi\ \langle\phi\rangle$ holds, we will construct a process called a *mirror*. We can think of this process as the most general environment of π that does not cause ϕ to fail. The mirror is a process whose outputs are the complement of the outputs of π. Outputs of the mirror must follow the specification ϕ. However, if process π causes ϕ to fail, the mirror may thereafter produce any output, since the blame for the failure falls to π. We can think of the mirror as a test generator for π: it can produce any input for π that satisfies the input assumptions of π.

Let $\pi = (I, O, T)$ be a process, and ϕ a specification. A π-failure of ϕ is any trace t such that $t \not\models \phi$ and the shortest prefix t' of t such that $t' \not\models \phi$ is π-terminal. Denote the set of π-failures of ϕ by *Failures*(π, ϕ). The ϕ-*mirror* of π, denoted M_ϕ^π is a process (I', O', T') where:

- $I' = O$,
- $O' = \mathcal{C} \setminus O$,
- $T' =$ *Failures*$(\pi, \phi)\ \cup \{t \mid t \models \phi\}$.

The following theorem shows how we can use a mirror process to test a triple $\langle\phi\rangle\ \pi\ \langle\phi\rangle$:

Theorem 3: Let ϕ be a specification and π a process. Then $\langle\phi\rangle\ \pi\ \langle\phi\rangle$ holds *iff* $M_\phi^\pi \parallel \pi \models \phi$.

Proof *Forward implication*. Suppose toward a contradiction that t is a trace of $M_\phi^\pi \parallel \pi$ and $t \not\models \phi$. Trace t must be a π-failure of ϕ (by the definition of mirror). Moreover, its projection onto the inputs and outputs of π is a trace of π (by the definition of composition). Trace t thus contradicts $\langle\phi\rangle\ \pi\ \langle\phi\rangle$. *Reverse implication*. Suppose toward a contradiction that there is a shortest π-terminal trace t of such that $t \not\models \phi$ and whose projection onto the inputs and outputs of π is a trace of π. This contradicts $M_\phi^\pi \parallel \pi \models \phi$. □

Because of the guarded command form of safety specification ϕ, the mirror process is not difficult to construct. The mirror process keeps track of the specification state. At each input or output event, it updates the state according to the update function τ. To produce an output event, it non-deterministically chooses an event on an output channel satisfying the guard γ. However, if an *input* fails to satisfy the appropriate guard, it goes to a special state from which all output events may be generated.

Theorem 3 gives us a way to test compositionally that a protocol implementation satisfies its specification ϕ. We decompose the system into locales. For each local ℓ, we need to verify the corresponding premise of Theorem 2. That is, letting $\pi = \| \ \ell$, we must verify $\langle\phi\rangle \ \pi \ \langle\phi\rangle$. To do this using Theorem 3, we construct the mirror process M_ℓ^π. We compose it with π to to obtain $M_\ell^\pi \parallel \pi$. We will call this process the *test process*. We then enumerate the traces of the test process and check that each satisfies ϕ. If this is true for all locales, we know that the system as a whole satisfies its specification. Of course, since the set of traces is in general infinite, this is not practical. Instead, we settle for checking a finite sample of the traces. The next section deals with how to generate this sample.

Example 6. Consider testing an implementation of our toy protocol. We start with server process B. The ϕ-mirror for B generates outputs that satisfy ϕ, so in particular these messages are always data-consistent. Since B simply echoes requests with responses, its outputs are also data-consistent, and moreover they always match previous requests. Thus, we can verify $\langle\phi\rangle \ B \ \langle\phi\rangle$. Consider client A. The client must guarantee data consistency of its outputs, but unfortunately this is impossible, since A cannot see all the requests of other processes. The test process generates, for example, the following trace that violates ϕ:

$$\textsc{Snd}_E(\textsc{Req}, 42, 0), \textsc{Snd}_A(\textsc{Req}, 42, 0)$$

In fact, our system does not satisfy its specification. The problem is that there are failures, such as the above, that cannot be blamed on any single process. In Subsec. V-B, we will see a solution to this problem.

III. Randomized testing

To sample the behaviors of the test process, we need to resolve the non-deterministic choices of this process in some way. There are several sources of non-determinism in the test process. First, the mirror has a choice among all output events satisfying the their action guard. We will call this *specification choice*. Second, if both the mirror and the process π under test are enabled to produce outputs, there is a choice of which process will execute next. We will call this *interleaving choice*. Third, process π may itself have internal non-deterministic choice.

There are many possible approaches to resolving the non-determinism. For example, we might try to systematically explore some finite subsets of the test process' behavior as in [20] or use white-box methods to improve coverage as in [5]. Our approach, however, is to use simple Monte Carlo sampling. In other words, we will replace non-deterministic choices with probabilistic choices. As we will see later, this may be as effective as more systematic approaches for complex Internet protocols.

For specification choice, we assume that a user supplies a target probability distribution over events. We have a distribution $p_\mathcal{C}$ over channels, and for each channel $c \in \mathcal{C}$, a distribution p_c over the parameter space R_c that gives the probability of a parameter valuation when the event label is c. Thus, the probability of an event (c, V) is $p_\mathcal{C}(c) \cdot p_c(V)$. Since the space of parameter valuations is not finite, there is no general notion of a uniform distribution. If a parameter ranges over the natural numbers, for example, we may choose an exponential distribution.

Suppose the mirror process is in a state defined by the Σ-structure $\mathcal{M}$. We can construct a characteristic set of constraints $\chi_\mathcal{M}$ whose unique model, up to first-order distinguishability, is $\mathcal{M}$. This is a reasonable assumption, since the reachable state structures are finite (though the *state space* is not).

The sampling problem is this: Given a Σ-action $\alpha = (\Lambda, \Sigma_a, \gamma, \tau)$ and a Σ-structure $\mathcal{M}$, draw randomly from the set of Σ_a-structures an assignment (valuation) V such that $\langle\mathcal{M}, V\rangle \models \gamma$. This is equivalent to drawing randomly from the models of $\chi_\mathcal{M} \wedge \gamma$, projected onto Σ_a. We define the ideal distribution over the models of χ as $p_\chi(\phi) = p_c(\phi|\chi)$. Thus, the probability of any parameter valuation V given a channel c is $p_c(V)/p_c(\chi)$ if $V \models \chi$ and zero otherwise.

A simple approach to draw from the ideal distribution would be to choose V randomly according to p_c, and then to reject any sample that does not satisfy χ (this is called rejection sampling). For typical protocol specifications, however, the rate of accepted samples may be negligible.

Unfortunately, in practice it is difficult to do better than rejection sampling if one wants to achieve a precise distribution. There exist precise methods to sample uniformly from clausal Boolean formulas, but they are costly [21]. MCMC methods are also used [22] but they require a large number of samples to converge to the desired distribution. This is not practical in our application, since the distribution is conditional on the specification state, which does not repeat. Thus, the cost of MCMC sampling cannot be amortized over a long sequence of samples.

Given that the choice of the ideal distribution is heuristic at best, and given the importance of achieving a high sampling rate in testing, it is more practical to think of the ideal distribution as a guide rather than a requirement. Here, we propose a simple and practical approach that uses the ideal distribution as a guide, applying an SMT solver. The method does not provide an approximation bound because we wish to trade off speed of sampling (and therefore of testing) for accuracy of the distribution. This point bears repeating: *it does not make sense to expend exponential computational resources to closely approximate an arbitrary distribution*, as this will

ultimately lower rather than increase the test coverage. For this reason, although the problem of conditional sampling with a precise distribution is well studied, we chose to take a heuristic approach. Having said this, while we show here that that our approximate approach is effective in practice, we do not explore the trade-off of distribution accuracy *vs* speed as it effects bug finding ability and leave this question for future research.

Sampling with SMT. While there is some degree of randomness in models produced by an SMT solver, this randomness is insufficient to produce an adequate diversity of test inputs, hence we do not rely on randomness in the SMT solver itself. Instead, to generate a sample, we first draw a parameter valuation from p_c. If the sample does not satisfy χ, instead of rejecting it, we apply an SMT solver that produces *UNSAT cores* to find a nearby sample that does satisfy χ. The algorithm for this is shown in fig. 1. We define a function SOLVERANDOM that given a constraint set χ and a vocabulary Σ, returns a satisfying assignment over Σ drawn roughly according to the ideal distribution p_χ. We start by drawing a sample from the base distribution p_c at line 3. This model is then translated into a characteristic set of constraints C using a function *StructureToConstraints*. We prefer to have fine-grained constraints. For example, if a symbol a represents an array of length n whose value in the model is $\mathbf{a}$, then we produce a corresponding set of characteristic constraints of the form $\text{select}(a, i) = \mathbf{a}[i]$ for $i = 0 \ldots n-1$. At line 5 we ask the SMT solver whether χ is consistent with the model constraints. If not, our sample did not satisfy χ. We consult an unsatisfiable core returned by function *UnsatCore*. This gives a heuristically small subset U of C that is inconsistent with χ. If U is empty, χ must be unsatisfiable, so we give up and return a special value $\perp$. Otherwise, we randomly choose one constraint from the UNSAT core to remove from C and repeat. When $\chi \wedge C$ is satisfiable, we return the projection ($\downarrow$) of the satisfying assignment onto Σ as our sample.

```
1  function SOLVERANDOM(χ,Σ)
2    //draw a model of χ over signature Σ, approximately distributed by p_χ
3    choose M ∈ structs(Σ) randomly, from distribution p
4    C ← StructureToConstraints(M)
5    while χ ∧ C is unsatisfiable:
6      U ← UnsatCore(χ,C)
7      if U = ∅ return ⊥
8      choose u ∈ U uniformly at random
9      C ← C \ {u}
10     choose M' s.t. M' ⊨ χ ∧ C
11   return M' ↓ Σ
```

Fig. 1. SOLVERANDOM: Randomize constraint solving with SMT and UNSAT core

In practice, this procedure produces a distribution of events that is far from ideal. It is fast, however, and we found it to generate a diversity of tests sufficient to uncover many protocol implementation errors that were not discovered by other means.

Finally, recall that there are two other sources of non-determinism in the test process: interleaving choice and internal choice of process π. For the former, we select which of the two processes to execute (π or its mirror) according to a fixed distribution. If the chosen process is not enabled to produce an output, we reject the sample and try again. In this scenario it is generally not possible to determine whether the process under test will eventually produce output. A pragmatic approach (also used in [9]) is simply to wait for a pre-determined time and if no output occurs, assume no output is enabled. For internal choice of π (for example, thread scheduling choice) we assume this is not under control of the tester and let the underlying system resolve the choices.

Randomized compositional testing has a key property that we will call *soundness*. That is, if the system violates the specification, then for some locale there is a local test that reveals the failure (by Theorems 2 and 3). Since the failing traces are finite, if the sampling distribution is everywhere non-zero, there is a finite probably of producing the failure. Thus in an infinite sequence of trials, we will eventually see the failure with probability one. We do not, of course, test infinitely. Nonetheless the fact that we do not rule out any failures is important heuristically, since it means that all failures are exposed to an adversarial test environment. Non-compositional approaches, such as MBT, do not have this property.

IV. DEPENDENT FIELD EXTRACTION

Modern SMT solvers are powerful tools. Unfortunately, they are not powerful enough to generate messages in a complex protocol such as QUIC at a speed sufficient for testing purposes. For example, we found that generating even a single, relatively simple, packet in the QUIC protocol required several minutes. This is because protocol states are large structures. If we feed all of the constraints in the characteristic formula of such a large structure into the SMT solver, the solver will choke. To avoid this problem, we need to write our guards in such a way that there are just a few essential random choices to be made, from which the remaining parameters can be derived deterministically. The random choices must depend on only a small part of the protocol state to avoid flooding the solver with too much data.

As an example, imagine that our protocol has messages called *frames*. Each frame has three fields: a beginning marker *bgn*, an ending marker *end*, and a payload array *pyld* that holds the segment of a long stream of bytes *data* that falls between these markers. We might write the guard of the *frame* action like this:

$$\begin{aligned} &\mathit{bgn} < \mathit{end} \wedge \mathit{end} < \mathit{len}(\mathit{data}) \wedge \mathit{len}(\mathit{pyld}) = \mathit{end} - \mathit{bgn} \\ &\quad \wedge \forall i.\ i < \mathit{len}(\mathit{pyld}) \Rightarrow \mathit{pyld}[i] = \mathit{data}[i] \end{aligned}$$

Suppose that *data* is part of the protocol state. Once we choose *bgn* and *end*, the value of *pyld* is determined. Moreover, the possible values of *bgn* and *end* depend only on the *length* of *data*. We could exploit these facts to optimize the solving process if they were more apparent in the form of the guard. Suppose we rewrite the guard like this:

$$\begin{aligned} &\mathit{bgn} < \mathit{end} \wedge \mathit{end} < \mathit{len}(\mathit{data}) \wedge \\ &\quad \wedge\ \mathit{pyld} = \mathit{segment}(\mathit{data}, \mathit{bgn}, \mathit{end}) \end{aligned}$$

Here, *segment* is a function interpreted in the solver's theory that returns a segment of an array. Now it is clear that we can use our SMT solver to solve the first two conjuncts, while ignoring the content of *data*. Then we can satisfy the last conjunct by simply computing *pyld* directly without using a solver.

In the remainder of this section, we present an algorithm that uses this idea to efficiently solve constraints in the appropriate form. We call we this approach *dependent field extraction*.

A. Modular constraint solving

The algorithm for constraint solving using dependent field extraction is shown in fig. 2. It takes as arguments a structure $\mathcal{M}$ (a state of the specification), a set Σ_a of symbols to solve for (parameters of an action) and a set of constraints C (the guard of an action) and returns an interpretation $\mathcal{M}_a$ of Σ_a such that $\langle \mathcal{M}, \mathcal{M}_a \rangle \models C$.

Informally, we say a term is *mutable* if we can modify its value without changing the truth value of formulae not referring to it. An example of a mutable term is an uninterpreted constant symbol v. An immutable term would be $x + y$, since its value cannot be modified without modifying the values of x or y.

To formally define mutability, we need to account for the presence of quantified logical variables. We say that a formula or term ϕ *refers* to a term t if some non-variable subterm of ϕ can be unified with t. For example, the formula $g(c) = d$ refers to the term $g(X)$ (where X is a variable) because we can unify $g(c)$ and $g(X)$ by replacing X with c. Given the constraint $g(c) = d$, we cannot define $g(X)$ independently, because the definition we choose might assign $g(c)$ a value other that d, so might be inconsistent with $g(c) = d$.

We write *Vars*(t) for the set of variables occurring free in term t and say that formula ψ preserves satisfiability of ϕ if satisfiability of ϕ implies satisfiability of $\phi \wedge \psi$.

Definition 2: A term t is *mutable* if, for all formulas ϕ not referring to t, every closed formula of the form $\forall$*Vars*$(t).(t = e)$, where e does not refer to t, preserves satisfiability of ϕ.

In other words, a mutable term can be defined arbitrarily while preserving satisfiability of ϕ, so long as ϕ does not refer to that term. Note that in the definitional formula $\forall$*Vars*$(t).(t = e)$, we require that e not refer to t so that the definition is not circular. As an example, if g is an uninterpreted function symbol and X is a variable, then $g(X)$ is mutable. Some terms with *interpreted* functions are also mutable. An important example is the term $f(v)$ where f is a *destructor* of an inductive datatype and v is a some uninterpreted constant. For example, say v is constant of a record type, and $f(v)$ represents the f field of record v. We wish to define $f(v) = e$ for some value e. We can start with any model of formula ϕ and change the value of the f field of v to e. If ϕ does not refer to $f(v)$ this leaves the truth value of ϕ unchanged. In particular, we will not change the evaluation of a term $f(w)$ representing the f field of record w.

We assume we have several procedures at our disposal. The procedure *Mutable* recognizes some syntactic class of mutable terms (for example, constants or destructors applied to constants). The procedure *UpdateStruct* takes a model of a set of constraints, and updates it so that $t = e$, where e is some value, provided t is mutable and not referred to in ϕ. For example, for a field reference $f(v)$, we set the f field of constant symbol v to e. The procedure *StructureToConstraints*, used also in SOLVERANDOM of Sec. III, converts a structure Σ to a characteristic formula.

Algorithm SOLVE in Fig. 2 first, calls a procedure *PartialEval* to partially evaluate the constraints in the given state interpretation Σ. This replaces any subterm that can be evaluated to a literal by its value. For example, in a structure that maps v to 0, the term $g(v+1)$ partially evaluates to $g(1)$. This is an important optimization because it can eliminate dependencies on large structures in the state. As an example, in a structure that maps a to the array $[0, 1, 2]$, the term *len*(a) partially evaluates to 3. Thus, partial evaluation may eliminate references to large data structures such as arrays. Then the algorithm searches at line 4 for a constraint c of the form $t = e$ where t is mutable. For simplicity, we only handle the case where t is a ground term. If t does not unify with any non-variable subterms of the remaining constraints or e, we call SOLVE recursively to solve the remaining constraints. We then *evaluate* term e in the resulting model $\mathcal{M}_a$ and update $\mathcal{M}_a$ so that $t = e$, using the procedure *UpdateStruct*. We know this can be done because t is mutable.

```
1  function SOLVE (M,Σ_a,C)
2      // find an interpretation M_a of Σ_a s. t. M, M_a ⊨ C
3      C ← PartialEval(M,C)
4      choose c ∈ C of the form t = e such that
5              Mutable(t) and ¬UnifiesWithSubterms(t, (C \ {c}) ∪ {e}):
6          M_a ← SOLVE (M,Σ_a,C \ c)
7          if M_a ≠ ⊥:
8              return UpdateStruct(M_a,t,Eval(M ∪ M_a,e))
9      else: // if there is no such c
10         χ ← StructureToConstraints(M ↓ vocab(C))
11         M_a ← SOLVERANDOM(C ∪ χ)
12         if M_a ≠ ⊥:
13             return M_a ↓ Σ_a
14     return ⊥
```

Fig. 2. Solve constraints using field extraction

On the other hand, if we do not find a constraint that we can extract, we continue, at line 10, to solve the full constraint set C. We remove from $\mathcal{M}$ the valuations of any symbols on which the constraints do not depend. Here, vocab(C) refers to the set of uninterpreted symbols in C. This may eliminate large data values such as the *data* array in our example. Then we convert Σ to a characteristic formula using the procedure *StructureToConstraints*. At line 11, we apply SOLVERANDOM to produce a model of this formula. Assuming it is satisfiable, we return the model projected on the action parameters Σ_a (at line 13.)

Consider again our example. Partial evaluation replaces the term *len*$(data)$ by a numeral (say, 42). The algorithm extracts the definition $pyld = segment(data, bgn, end)$ because *pyld* is mutable and not referred to in the remaining constraints $bgn <$

end $\wedge$ *end* $<$ 42. It then recurs on these constraints. Since this removes the last dependence on *data*, its interpretation is removed from the state structure. Now the constraint can be easily solved. Returning from the recursive call, we then evaluate the term *segment*$(data, bgn, end)$ and update *pyld* to this value. This approach can result in a large improvement in solving performance.

V. Testing pragmatics

In this section, we consider some additional issues that arise in applying NCT in practice.

A. The test shim

Thus far, we have tacitly assumed we have a way to compose the mirror process with the collection of processes in a locale. In practice, this may be a non-trivial problem. The processes in the locale consist of software, expressed in some programming language and interpreted by a physical machine, likely with the assistance of a good deal of additional software, including a language run-time and an operating system. We need some way to translate between the abstract events of our specification and real occurrences in this physical system (for example, instances of API calls, or signals on a wire). We will call this translation process a *shim*.

One approach to making a shim would be to replace the run-time environment of the software components being tested by a virtual or simulated environment. For example, suppose the software is written in C and communicates over the network using calls to the Unix sockets API. We could simulate these calls with calls into the shim that would translate send calls into abstract send events, and abstract receive events into the return values of receive calls. There are significant difficulties in this approach, however. For one, implementations may be written in different languages and use different system API's. This means we may need a different shim for each implementation.

A much simpler approach would be to test the software through the real physical network. In this approach the shim handles receive events of a process by sending it a message over the network, and interprets messages received over the network from the process as send events of that process. It is reasonable to ask, however, whether this testing procedure is sound. That is, is it not possible that delays or duplications introduced by the physical network (or a loop-back interface in the operating system) will mask a protocol violation, or perhaps cause one?

We will argue informally that, under reasonable assumptions, compositional testing *via* the network is sound. We assume a network model in which the network guarantees only not to invent messages. We can think of this network as a composition of two groups of processes, reception processes denoted by Rcp_i, and delivery processes denoted by Dlv_i. For each input channel Inp_i of the network, the reception process Rcp_i inputs messages on Inp_i and outputs them on a *commit channel* $Commit_i$. For each output channel Out_i of the network, the delivery process Dlv_i inputs messages from *all* of commit channels and output them to Out_i. See Fig. 3 for an illustration.

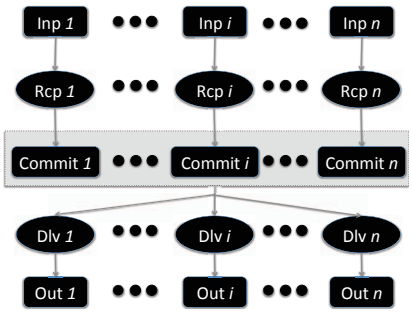

Fig. 3. The reception and delivery processes

The reception and delivery processes, like the network as a whole, promise only not to invent messages. The network modeled in this way is externally indistinguishable from the network model without the commit events.

Suppose that instead of using ϕ to specify the message send events of our processes, we specify the imaginary commit events. When we decompose the system into locales, we include in the locale of each process its corresponding reception and delivery processes. Our test process is the same as before, except that now the mirror is communicating with the test process through processes that arbitrarily delay and duplicate messages, in other words, processes that act like a network. Testing through the physical network can be thus be viewed as simulating the imaginary reception and delivery processes in our network model. The physical network will not, of course, simulate all possible reorderings of messages allowed by the model. However, by adding random reorderings in the network we can simulate all possible message arrival orders with non-zero probability and thus maintain soundness. Provided we specify the protocol in a way that allows for arbitrary delays and duplications, it is sound to test implementations of the protocol *in situ*, through a physical network or operating system. This greatly simplifies the problem of testing many implementations of the same protocol.

B. Background assumptions

Often, in specifying of a protocol, we must make some assumptions that are not justifiable within the model we are using. An example of this is *symmetry breaking*. To establish identities of processes on a network, or to avoid deadlocks, we may assume that processes have access to some source of randomness and that two processes will make the same random choices with negligible probability (e.g., [23], [24] and IEEE 1394.). We cannot *prove* that two processes do not choose the same nonce value. Rather, we wish to ignore system executions in which they do.

Such assumptions do not fit into our compositional framework because, when they fail, there is no single process that we can blame. For this reason, we will augment our compositional rule with a *background assumption* β. We can express the idea

that a process ϕ does not cause ϕ to fail without breaking the background assumption by this triple:

$$\langle \beta \wedge \phi \rangle \ \pi \ \langle \beta \Rightarrow \phi \rangle$$

The implication on the right allows π to produce any output that violates β, because we do not care about such traces. The new rule for closed systems is:

$$\frac{\text{for all } \ell \in L: \quad \langle \beta \wedge \phi \rangle \ \parallel \ell \ \langle \beta \Rightarrow \phi \rangle}{\parallel \Pi \models \beta \Rightarrow \phi} \tag{4}$$

That is, if a locale can only cause the property to fail by violating the background assumption, then all traces of the composition that do not violate the assumption must satisfy the property.

Example 7. Recall Ex. 6 in which we could not rule out requests from different processes having the same connection id. Obviously, no single client process can enforce this requirement, since the client cannot see the messages of all other clients. Rather, we assume that the connection ids are large random numbers, and thus we wish to ignore traces in which two clients choose the same connection id. We can express this assumption as a safety property with a state relation U that records the set of pairs (c, N) such that nonce N has been used on channel c. The background assumption β is expressed using the following actions:

$$\begin{array}{lll} \textsc{Snd}(M,N,D): & (\forall c.(U(c,N) \Rightarrow c = \hat{p})) & \\ & \quad \rightarrow \quad \{U(\hat{p},N) := \mathsf{true}\} & \\ \textsc{Rcv}(M,N,D): & \mathsf{true} \quad \rightarrow \quad \{\} & \end{array}$$

Here, we use the special symbol $\hat{p}$ that represents the channel on which the event occurs. Thus, this specification is not process-oblivious. With this β, the mirror process of a client is prevented from generating connection ids that conflict with the client, and the checker ignores cases where the client produces a conflicting id. Thus, we can prove compositionally that the clients and servers together implement protocol specification ϕ under assumption β. □

VI. Specifying and Testing QUIC

We now illustrate the application of the NCT methodology described above using a case study on the QUIC transport protocol. Here, we describe QUIC and the case study only to the extent necessary to illustrate the application of the various aspects of the methodology, such as locales and the dependent fields optimization. A full account of the QUIC case study can be found in [25].

The case study was designed to test several hypotheses:

(A) that Internet protocols such as QUIC require a testing regimen that is adversarial and checks protocol compliance;
(B) that this can be provided effectively by a specification-based testing approach using a safety specification, and
(C) that a suitable specification can be distilled in practice from standards documents and compositional testing of a collection of implementations.

An explicit non-goal of the study is to produce a full or complete specification of QUIC. The function of the specification in the case study is to aid in finding bugs in implementations of QUIC.

A. An introduction to QUIC

QUIC can be thought of as a stack of protocols, each of which provides one aspect of the overall transport service. At the bottom of the stack, UDP provides datagram services. Above this the *packet protection layer* provides secrecy by encrypting QUIC packets, which are encapsulated into UDP datagrams. Above this, the *packet protocol* provides loss detection using sequence numbers. The *frame protocol* provides (among other things) ordered stream data. Sequences of frames are encapsulated in packets. Each data frame carries a stream identifier, a sequence of bytes, and the offset of those bytes within the stream. This allows the stream data to be reconstructed at the receiving end in spite of datagram reordering and supports multiple independent streams. Above the frame protocol, the *security handshake protocol*, which is a modified version of TLS 1.3, exchanges handshake messages using the frame protocol. Once a shared secret has been established by the handshake protocol, keys can be derived for encryption and decryption by the protection layer. Finally, at the top is the *application* layer in which peers send and receive reliable, secure, authenticated data streams. See Figure 4.

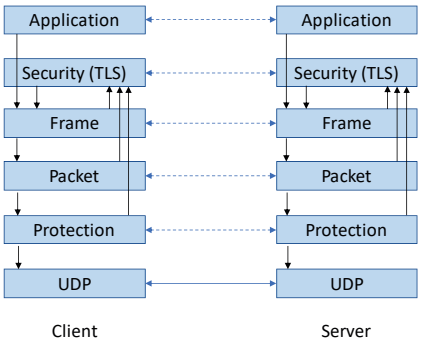

Fig. 4. QUIC protocol layers. Arrows represent dependencies between layers.

The basic unit of communication in QUIC is the *connection*. A connection is a point-to-point channel that provides multiple independent data streams. A connection is established when a client sends an *initial packet* to a server. This packet provides a *connection identifier* (CID) to the server, a string of bytes that uniquely identifies the connection. It also contains a frame with the first security handshake message. The server responds with its own initial packet, containing the server's CID and the second handshake message. Subsequent *handshake packets* are protected with handshake keys derived from the initial messages. Once the handshake is complete, session keys are available and transmission of session data commences. QUIC packet types are thus partitioned into four *encryption levels* using different keys: initial, handshake, 0-RTT (for early data) and 1-RTT (for normal data with forward secrecy).

In addition to CID's, QUIC packets contain unique sequence numbers that are used to detect packet loss. A peer sends acknowledgment (ACK) frames to indicate packet sequence numbers that it has received. A packet that is not acknowledged is considered lost after some time. Rather than retransmit the packet, the peer retransmits its frames, as needed, in subsequent packets with different sequence numbers. Clients can migrate to new network addresses. Before using the new address, the server validates that the client actually controls the new address using special frames. Additional CID's can be issued to prevent connection tracking by attackers. The draft protocol, as of version 17, has 20 frame types that are used for various purposes: data transmission, loss detection, flow control, connection state management, management of CID's and so forth.

B. A formal safety specification for QUIC

Our formal safety specification for QUIC is written as a collection of actions in the Ivy language [26]. These actions describe events at each layer of the protocol. This decision was crucial for performance of test generation. That is, in principle, we could have specified only the UDP datagrams that appear on the network (or more properly, their cleartext content). However, generation of these events using an SMT solver would be too slow for testing purposes. By breaking large events into smaller events at higher protocol layers and applying dependent field extraction, we make test generation practically feasible.

The most interesting actions are at the packet and frame layer. The packet action has three parameters: the source and destination endpoints (IP address and port number) and the packet content. The packet content is expressed as a record having various fields representing the encryption level, source and destination CID's, sequence number and payload, the last represented as an array of frames carried by the packet. The guard of the packet action specifies, for example, correct use of CID's and sequence numbers. It also defines the payload field to be a sequence of frames enqueued by the frame protocol for the given encryption level. That is, these records are communicated from the frame protocol to the packet protocol *via* the specification state. This is one example of the use of dependent field extraction. By delegating the production of frames to separate frame protocol actions and defining the packet payload appropriately, we avoid sending the actual frames to the SMT solver. To use this optimization, it was crucial to specify events at multiple layers. The update function of the packet action records various needed history information, such as which sequence numbers have been used and which frames have been transmitted in packets.

Each frame type in the frame protocol is a specified by a corresponding action. As an example, a frame action of STREAM type carries application data. The parameters of this action are the frame content, the source and destination CID's and the encryption level. The stream frame record contains a stream id field as well as length and offset fields indicating that range of data within the stream that is begin transmitted. The guard contains a variety of requirements. For example, the source and destination CID's must be connected, and the encryption level must be '1-RTT' (that is, STREAM frames must be sent only with 1-RTT encryption). We also require that the necessary encryption keys have been generated by the security layer. Crucially, the data bytes contained in the frame must match the corresponding bytes at the application layer. We omit a variety of other requirements here. The update of the action enqueues the frame for eventual transmission in a packet and performs and stores some additional history information.

Another use of dependent fields occurs in the STREAM frame action. As in Sec. IV, we define the data in the frame as a segment of the application data in the protocol state. This prevents this large array from slowing the SMT solver and allows our tester to fetch HTML files from the real server (and allows the corresponding client tester to serve files).

The protocol layer actions behave like interleaving parallel processes, sharing data through the specification state. For example, the security handshake protocol exchanges messages with the frame protocol via shared state symbols and also shares computed cryptographic secrets with the protection layer.

Overall, our current formal safety specification of the QUIC wire protocol consists of 52 data type declarations, 45 state symbols (the functions and relations in the state signature Σ) and 30 actions. The data type declarations are primarily record types that represent packets, frames, TLS messages and the like, as well as basic types for entities like CID's and sequence numbers. The state symbols carry a wide variety of protocol history information, for example, the association between client and server CID's, the used and acknowledged sequence numbers for each connection, the IP addresses used by the client, the handshake and application data transmitted, the status of streams, the flow control parameters and so on.

C. Creating the test process

Given a specification expressed as a collection of actions, the Ivy tool can generate a randomized mirror process. This process is expressed in the C++ language and use the Z3 SMT solver [27] with Algorithm SOLVERANDOM to generate events consistent with the specification.

The handshake protocol illustrates a convenient use of locales, as described in Subsec. II-C. In particular, because we lacked a sufficiently detailed formal specification of the TLS 1.3 protocol from which to generate events at the security layer, we instead used an actual implementation of TLS 1.3 [28] as a self-specifying process. For example, when testing a server implementation, the instances of TLS on the client side are considered part of the locale, and are executed concretely. This sacrifices some generality of testing, since the concrete implementation does not generate all possible TLS behaviors. On the other hand, it helps greatly to circumscribe the formal specification effort. The ability to use an off-the-shelf TLS 1.3 implementation was an important enabling factor in testing QUIC. In testing clients, we use a background

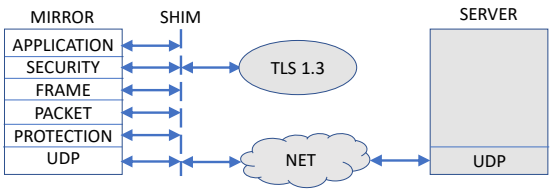

Fig. 5. Structure of the test process for QUIC servers. Processes in the locale are in grey. The shim connects the mirror to the locale and also infers hidden server events at higher protocol layers.

assumption (Subsec. V-B) to filter out cases where the client violates the specification by choosing the same nonce CID as another client.

The shim that connects the mirror process to the concrete software under test is also written in the Ivy language. We take the approach described in Subsec. V-A of testing through the physical network so that we can easily test a variety of implementations of QUIC. The shim is also used to connect security layer events in the mirror with concrete API calls of the TLS 1.3 implementation. The shim has one further task. That is, the concrete protocol implementations do not expose protocol events above the UDP layer. Instead of instrumenting the implementation code to output these events, we *infer* them in the shim by inspecting the packets on the wire. That is, a UDP datagram on the wire allows us to infer the hidden events that must have occurred in the past at the higher layers. This inference is straightforward, and is accomplished with additional 28 lines of Ivy code.

As noted in Sec. III, the user must choose an appropriate target distribution for sampling out of the space enabled events. The Ivy tool provides some primitive mechanisms for doing this. To obtain a diverse sampling of behaviors of the implementation, it is important to choose this distribution carefully. For example, we lowered the probability of certain events, such as CONNECTION_CLOSE frames, that tend to cancel all activity. We also limited the range of connection ids, since choosing a different connect id for each packet would result in no progress of the protocol.

The overall structure of the test process is depicted in Fig. 5. This shows how the mirror process generated by Ivy is connected to the various processes in the test locale, via the shim. Interacting with the protocol implementation over the network, the mirror can generate protocol events at rate of approximately 10Hz (or < 0.01 Hz without dependent field extraction).

D. Results

We now consider the results of the interactive process of developing the protocol specification and testing implementations against it. We applied the testing methodology to HTTP servers based on the four chosen implementations of QUIC.

We wrote an initial version of the specification based on draft versions of the QUIC protocol standards documents [29]. We then tested the various server implementations against this specification. After analysis, failures of these tests often indicated errors in the implementations or the draft standard, but sometimes also indicated errors in our safety specification. Consultation with implementers was often required to determine this. Failure of a guard in checking the process output sometimes indicates that the guard is in fact too strong, and that the output should have been allowed. On the other hand, failure of a process to handle an input (for example, by flagging a protocol error) sometimes indicates the that guard that produced that input in the mirror is too weak and needs to be strengthened to rule it out. Because testing revealed when the specification is either too weak or too strong, we were able to use it to develop an adequate specification for testing QUIC implementations, confirming hypothesis *C*. In particular, we found that it was challenging but possible to keep up with successive draft versions of the protocol as they were developed.

In addition to protocol compliance errors, we recorded crashes of the servers and failures to make progress, which we defined as an anomalously low rate of data transfer during a test. Over the course of approximately four engineer weeks of testing, this revealed a total of 27 errors in the implementations. We also determined, where possible, the root cause of the errors, and classified the reason for detection of the errors. These categories are 'adverse stimulus', meaning the the randomized tester produced an unexpected message order or parameter value, and 'compliance violation', meaning that we detected the error because we monitored the trace with a formal specification. Of the 23 errors to which we were able to assign root causes, all were classified as being either due to 'adverse stimulus' (78%) or to 'compliance violation' (57%) or both. None were detected by previous directed testing or interoperability testing. This confirms hypothesis *A*: it is important both to test compliance to a common standard, *and* to test in an adversarial environment. We also found that four of the errors were caused at least in part by ambiguities or contradictions in the draft RFCs and four were possibly exploitable by an attacker (apart from crashes, which might also be exploitable in various ways).

This also illustrates an important point about specifications: a formal specification of a highly complex system need not and perhaps cannot be fully complete (in the sense of saying everything about the system that we want to say) or even fully correct. We should judge a specification by the extent to which it serves its function. In this case the function is primarily to expose errors *via* testing. Our specification of QUIC is limited to safety properties and leaves some protocol aspects unspecified, but it is sufficient to interact with real servers to transfer files on the network without the servers detecting protocol errors. Moreover, it is strong enough to detect numerous protocol violations, confirming hypothesis *B*.

We give two examples of implementation errors we detected that represent vulnerabilities. One of these is a possible denial-of-service (DoS) attack by an off-path attacker. We discovered a trace in which a server ceased at some point to send any packets (representing a progress failure). Further analysis revealed that this was caused by a rapid switching of the client

IP address between two values, which might be simulated by an attacker. This was determined to be a weakness not just in the implementation, but also in the standard, which was subsequently modified to mitigate such attacks. This error was detected only because the source IP address was randomized by the tester, producing an adverse stimulus. Another vulnerability we detected is a data leak in one implementation, similar to the "heartbleed" vulnerability discovered in SSL/TLS [30]. This resulted in arbitrary server memory contents being sent to the client, and was detected when a server sent incorrect bytes in a retransmission of a stream frame, violating the specification. The apparent cause of this was adverse use of flow control by the randomized tester. It is interesting to note that, although we tested only for violation of safety properties, a number of security vulnerabilities emerged from the tests. One interpretation of this is that our tester, while producing only legal protocol behavior, nonetheless produced a wide variety of unexpected stimulus. Pushing the boundaries of the protocol is in a sense form of attack, and a difficult one to detect.

VII. Future work

Though it is clear that the randomized NCT approach is effective in exposing previously unseen behaviors of implementations, there are many ways in which it could potentially be improved to better explore the space. For example, one could "fuzz" randomly generated traces to produce new traces, perhaps using code coverage as a heuristic. This could be done using white-box methods, as exemplified by the KLEE tool [5]. In these methods, SMT solvers are used to discover inputs that drive the implementation along different code paths. To date, white-box methods have not been very effective in producing deep errors that require long exchanges of messages in Internet protocols [8], [31]. However, they might be an effective adjunct to randomized specification-based testing. Another promising approach is grey-box testing [32]. This method essentially performs a random walk in the space of inputs starting from certain "seed" inputs. Information about covered paths in the implementation is used to bias the walk. Traces produces by NCT could be used as such seeds. While we do not believe that code coverage is a valid measure of the quality of a test regime, we do expect that coverage information can be used effectively as a heuristic guide in NCT to more effectively find bugs.

NCT has the property that it can generate the long legal sequences of messages that are needed to go deep in the state space of the implementation. We think this property could be exploited in many applications, especially in security. That is, any technique that is used to search for vulnerabilities or exploits could potentially benefit from having a rich diversity of starting states. The fact that we accidentally discovered some vulnerabilities in QUIC and its implementations is evidence of this. We intend to explore this possibility in the future. We are also interested in incorporating specific attacker models into the methodology, and in the question of how to generalize accidentally discovered attack traces into attack strategies that can be replayed.

VIII. Conclusions

In this paper we developed a NCT, a network-centric methodology for compositional testing. NCT enables one to apply modular assume-guarantee reasoning principles to the development of Internet protocols, and to validate implementations through adversarial testing. Unlike previous compositional approaches, NCT does not use local specifications of processes. Rather, it is based on single global specification of the protocol and a notion of causality of failures that assigns blame for the failure of the specification to a single process. This allows us to effectively generate tests for protocols with an unbounded number of participants, and to monitor specifications on a network. For each process, the global specification serves as both an assumption and a guarantee. This makes it possible to automatically generate randomized test environments that produce adverse stimulus and, at the same time, check the implementation's responses for specification compliance. This fulfills the need we identified to test implementations against a common formal specification, in an adversarial manner. Moreover, because all assumptions of one process are guarantees for another, NCT allows to detect when the specification is too weak and to refine it (in fact, we did this many times with QUIC). Thus, we can apply the NCT methodology to distill formal specifications from the protocol knowledge embodied in the implementations.

To permit randomized generation of complex protocols such as QUIC, we developed pragmatic techniques for randomized constraint solving that approximate a desired distribution, and a method of decomposing the solution process called 'dependent field extraction' that makes it possible to generate protocol traffic in the case of deeply nested message structures with large data transfers. To enable this, we divided the specification of QUIC into actions at the various protocol layers. This methodology allowed us to test the complete QUIC stack as a black box, a necessity in the case of implementations that are not cleanly layered.

We saw in the case of QUIC that adverse stimulus generated by the randomized tester, combined with monitoring by the formal specification, was very effective in detecting errors in the implementations that were not detected by other means. As a side effect, we identified vulnerabilities in both the implementations and the standard, and exposed cases of ambiguities and contradictions in the standard that have been remediated. This shows clearly that a formal specification is a useful tool in developing and implementing a network protocol.

As we noted, the SSL/TLS ecosystem suffered many difficulties owing to the lack of compliance of implementations in the wild to an unambiguous common protocol specification. Our hope is that providing such a specification in a *testable* form will be a step in preventing such difficulties in QUIC, which is planned as the future basis of the World-Wide Web. We also hope that the form of the specification is simple enough that future developers of QUIC can use it as

a reference, though this remains to be seen. In general, we see this work as a step in the process of integrating formal specifications as a complement to other approaches in Internet standardization.

REFERENCES

[1] T. M. Corporation, "CVE-2014-3566," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3566, 2014.

[2] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of TLS," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 2015, pp. 535–552. [Online]. Available: https://doi.org/10.1109/SP.2015.39

[3] I. Ristic, "POODLE bites TLS," December 2014.

[4] H. Lee, J. Seibert, D. Fistrovic, C. E. Killian, and C. Nita-Rotaru, "Gatling: Automatic performance attack discovery in large-scale distributed systems," *ACM Trans. Inf. Syst. Secur.*, vol. 17, no. 4, pp. 13:1–13:34, 2015.

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008, pp. 209–224.

[6] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P. Strub, "Implementing TLS with verified cryptographic security," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE, 2013, pp. 445–459.

[7] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook, "Continuous formal verification of amazon s2n," in *Computer Aided Verification - 30th International Conference Part II*, vol. 10982. Springer, 2018, pp. 430–446.

[8] F. Rath, D. Schemmel, and K. Wehrle, "Interoperability-guided testing of QUIC implementations using symbolic execution," in *Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ 2018)*. ACM, December 2018, pp. 15–21.

[9] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, ser. Lecture Notes in Computer Science. Springer Verlag, January 2008, vol. 4949, pp. 39–76.

[10] J. Paris and T. Arts, "Automatic testing of TCP/IP implementations using quickcheck," in *Proceedings of the 8th ACM SIGPLAN Workshop on Erlang, Edinburgh, Scotland, UK, September 5, 2009*. ACM, 2009, pp. 83–92.

[11] J. Bozic, L. Marsso, R. Mateescu, and F. Wotawa, "A formal TLS handshake model in LNT," in *Proceedings Third Workshop on Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation, MARS/VPT@ETAPS 2018, and Sixth International Workshop on Verification and Program Transformation, Thessaloniki, Greece, 20th April 2018.* To be published in EPCT, 2018, pp. 1–40.

[12] B. Neelakantan and S. V.Raghavan, "Protocol conformance testing – a survey," in *Computer Networks, Architecture and Applications*, S. V. R. et al., Ed. Springer, 1995, ch. 1, pp. 175–191.

[13] S. Bishop, M. Fairbairn, H. Mehnert, M. Norrish, T. Ridge, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the sockets API," *JACM*, vol. 1, no. 66, pp. 1–77, 12 2018.

[14] J. Misra and K. M. Chandy, "Proofs of networks of processes," *IEEE Trans. Software Eng.*, vol. 7, no. 4, pp. 417–426, 1981.

[15] K. A. Elkader, O. Grumberg, C. S. Pasareanu, and S. Shoham, "Automated circular assume-guarantee reasoning," *Formal Asp. Comput.*, vol. 30, no. 5, pp. 571–595, 2018.

[16] D. Giannakopoulou, C. S. Pasareanu, and C. Blundell, "Assume-guarantee testing for software components," *IET Software*, vol. 2, no. 6, pp. 547–562, 2008.

[17] K. L. McMillan, "Modular specification and verification of a cache-coherent interface," in *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*. IEEE, 2016, pp. 109–116.

[18] W. P. de Roever, H. Langmaack, and A. Pnueli, Eds., *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September 8-12, 1997. Revised Lectures*, ser. Lecture Notes in Computer Science, vol. 1536. Springer, 1998.

[19] O. Grumberg and D. E. Long, "Model checking and modular verification," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 843–871, 1994. [Online]. Available: https://doi.org/10.1145/177492.177725

[20] M. Musuvathi and S. Qadeer, "CHESS: systematic stress testing of concurrent software," in *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers*, ser. Lecture Notes in Computer Science, G. Puebla, Ed., vol. 4407. Springer, 2006, pp. 15–16. [Online]. Available: https://doi.org/10.1007/978-3-540-71410-1_2

[21] K. S. Meel, M. Y. Vardi, S. Chakraborty, D. J. Fremont, S. A. Seshia, D. Fried, A. Ivrii, and S. Malik, "Constrained sampling and counting: Universal hashing meets SAT solving," in *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016.*, ser. AAAI Workshops, A. Darwiche, Ed., vol. WS-16-05. AAAI Press, 2016. [Online]. Available: http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12618

[22] N. Kitchen and A. Kuehlmann, "A markov chain monte carlo sampler for mixed boolean/integer constraints," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 446–461. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_34

[23] D. Lehmann and M. Rabin, "On the advantages of free choice: A symmetric and fully distibuted solution to the dining philosophers problem (exended abstract)," in *POPL'81*, 1981, pp. 133–138.

[24] A. Itai and M. Rodeh, "Symmetry breaking in distributed networks," *Inf. Comput.*, vol. 88, no. 1, pp. 60–87, 1990.

[25] K. L. McMillan and L. D. Zuck, "Formal specification and testing of QUIC," in *Proc. ACM Special Interest Group on Data Communication (SIGCOMM19)*. ACM, 2019, to appear.

[26] K. L. McMillan, "Ivy," http://microsoft.github.io/ivy, Last updated 2019.

[27] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[28] P. T. team, "picotls," https://github.com/h2o/h2o/tree/master/deps/picotls, 2019.

[29] Internet-Draft, "QUIC: A UDP-based multiplexed and secure transport (version 18)," https://tools.ietf.org/id/draft-ietf-quic-transport-18, 2019.

[30] T. M. Corporation, "CVE-2014-0160," 2014. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1060

[31] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. D. Millstein, "Analyzing protocol implementations for interoperability," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*. USENIX Association, 2015, pp. 485–498. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pedrosa

[32] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Trans. Software Eng.*, vol. 45, no. 5, pp. 489–506, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2017.2785841