Formally Discovering and Reproducing Network Protocols Vulnerabilities

Christophe Crochet, John Aoga, and Axel Legay

INGI, ICTEAM, Université catholique de Louvain, Place Sainte Barbe 2, L05.02.01, 1348 Louvain-La-Neuve, Belgium
{firstname.lastname}@uclouvain.be

Abstract. The rapid evolution of cyber threats has increased the need for robust methods to discover vulnerabilities in increasingly complex and diverse network protocols. This paper introduces *Network Attack-centric Compositional Testing* (*NACT*) [12], a novel methodology designed to discover new vulnerabilities in network protocols and create scenarios to reproduce these vulnerabilities through attacker models. *NACT* integrates composable attacker specifications, formal specification mutations, and randomized constraint-solving techniques to generate sophisticated attack scenarios and test cases. The methodology enables comprehensive testing of both single-protocol and multi-protocol interactions. Through case studies involving a custom minimalist protocol (`MiniP`) and five widely used `QUIC` implementations, *NACT* is shown to effectively identify, reproduce, and find new real-world vulnerabilities such as version negotiation abuse. Additionally, by comparing the current and older versions of these `QUIC` implementations, *NACT* demonstrates its ability to detect both persistent vulnerabilities and regressions. Finally, by supporting cross-protocol testing within a black-box testing framework, *NACT* provides a versatile approach to improve the security of network protocols.

Keywords: Formal Specifications, Formal Verification, Mutation Testing, Network Attacks, Systems, Internet protocols, `QUIC`, Concrete Implementation, Adverse Stimuli, Framework

1 Introduction

In today's hyperconnected world, network protocols serve as the foundation of global communication infrastructures, yet they remain vulnerable to sophisticated cyber threats. Protocol vulnerabilities provide a fertile ground for attackers to disrupt services, steal sensitive information, and cause widespread damage, as seen in high-profile incidents such as the *Heartbleed* [37] and *Log4Shell* [15] attacks. The unpredictability of cyberattacks lies in their ability to exploit weaknesses in ways not anticipated during the protocol's design. Attackers frequently leverage *zero-day* vulnerabilities and subtle deviations from formal specifications to create unexpected behaviors that lead to breaches [40]. This challenge

highlights the necessity of anticipating attacks by uncovering potential vulnerabilities before they can be exploited [38]. Testing and simulation approaches play a crucial role in this proactive defense strategy.

Previous studies have delved into various fault-based testing methodologies. Some methods focus on validating protocols according to their defined specifications (conformance testing) [16,32], while others target known vulnerabilities directly [35,36], especially through mutation testing [41], which creates test scenarios by injecting minor controlled errors (mutants) into protocols to gauge their resilience [33]. These strategies typically fall under the umbrella of *black-box* or *fuzz testing* [36]. It is crucial to perform tests directly on implementations to enhance the realism and applicability of these tests for real-world network situations. This is where *Model-Based Testing (MBT)* [28] comes in, using an advanced language to describe protocol specifications and validate them directly against real implementations. This method facilitates the identification of potential code vulnerabilities that an attacker can exploit in practical scenarios, especially given the growing complexity of modern protocol implementations.

Recently, *Network-centric Compositional Testing* (*NCT*) [23] has emerged as a pioneering MBT approach [28] that uses formal protocol specifications, implemented with tools such as *Ivy* [29], to generate test tools automatically. *NCT* has effectively identified bugs and vulnerabilities, particularly in real-world protocols such as `QUIC`.

However, *NCT* focused on individual compliance checks and did not address the dynamic and diverse nature of actual network interactions, which are essential to detect potential vulnerabilities before they can be exploited.

To bridge this gap, we present Network Attack-centric Compositional Testing (*NACT*), a novel *model-based mutation testing* that builds upon *NCT*. *NACT* is tailored to thoroughly evaluate real-world network protocols for security flaws. Using compositional attacker models, mutations in formal specifications, and randomized constraint-solving techniques, *NACT* can uncover previously unknown vulnerabilities by emulating various advanced attack scenarios. Our method facilitates the examination of multi-protocol and cross-protocol interactions in black-box settings, thereby creating realistic, adversarial circumstances reflective of potential exploits in live environments. The capability to conduct multi-protocol testing and multipoint communications is becoming increasingly crucial with the rise of microservice architectures and distributed systems. We validate our method's effectiveness by testing several recent vulnerabilities in the widely used `QUIC` protocol, successfully identifying new significant security weaknesses, including version negotiation issues and buffer overflows.

The remainder of the paper is organized as follows. Section 2 offers an overview of network protocol vulnerabilities, associated attacks, the *NCT* methodology, and its inherent limitations. Section 3 introduces the *NACT* methodology, elaborating on the mutations of formal specification, their implementation, and integration within a compositional testing framework. Section 4 discusses our case studies on five `QUIC` implementations, showcasing the ability

of *NACT* to identify complex vulnerabilities. Finally, Section 5 concludes and suggests possible directions for future work.

2 Background

This section covers the core concepts of network attacks, how to test network protocols against these threats using network-centric approaches, and the limitations of these approaches, emphasizing the need for more comprehensive testing frameworks.

(1) Network Attack Vectors and Agents. In network security, a **vulnerability** is a defect or weakness present in the design, implementation, or operation of a system that an attacker can exploit to breach the system's security attributes, such as *confidentiality*, *integrity*, *availability*, and *authenticity*. When a malicious entity discovers and exploits a vulnerability, it results in a **network attack**, which is any intentional action to disrupt standard system functioning.

(a) Types of Network Attacks. To understand the different forms of network attacks, we conducted a literature review and identified the most prevalent types of attacks that can occur in network environments. We classify these attacks according to their mode of operation. Table 1 summarizes the general categories of network attacks as referenced in various works [8, 10, 18, 21].

Type of Attack	**Description**
TLS/SSL Connection Attacks	Exploiting vulnerabilities in the TLS handshake process, such as protocol downgrade or renegotiation attacks, to intercept, decrypt, or alter encrypted communication data.
Injection	Injecting malicious data/code into a system or application to manipulate its operations.
Eavesdropping	Intercepting communications between two parties without their consent.
Service Disruption	Overloading or disrupting services to make them unavailable to legitimate users.
Spoofing	Impersonating a legitimate entity to deceive users or systems.
Protocol-Based Attacks	Exploiting inherent weaknesses or misconfigurations in communication protocols (e.g., TCP Slowloris) to overwhelm, disrupt services, or gain unauthorized access.

Table 1: Network attacks categories from the literature.

These attack types form the basis for our testing framework, allowing us to simulate various attack scenarios and validate the robustness of network protocols under different adversarial conditions.

(b) Attack Vectors. An **attack vector** is the specific method or pathway that an attacker uses to exploit a vulnerability in a system. Understanding attack vectors is essential in constructing effective security models and testing methodologies, as they directly influence how attacks are simulated and mitigated. We identified three main attack vectors relevant to network security testing, presented in Table 2. Although malicious clients and servers might appear similar, the directions of their interactions differ. A malicious client begins communication and strays from protocols to perform attacks. In contrast, a malicious server responds to requests with harmful actions, taking advantage of client trust. The impact of a malicious client is precise and targeted, whereas a malicious server can influence numerous users and disrupt the entire network, magnifying the damage. A special case of this is the Man-in-the-Middle (MitM) attack, where the attacker intercepts and manipulates the communication between a client and server, often impersonating both simultaneously. This dual role allows the attacker to perform activities such as spoofing, eavesdropping, and data injection without the communicating parties' awareness.

Attack Vector	**Description**	**Attack Surface**	**Type of Attack**
Man-in-the-Middle (MitM)	Intercepts and potentially alters the communication between two parties without their knowledge, compromising message integrity and authenticity.	Network communication between client and server	Eavesdropping Injection TLS/SSL Connection Protocol-Based Attacks Service Disruption Spoofing
Malicious Client	Behaves as a legitimate user but intentionally violates protocol rules to disrupt the system or steal data.	Interaction with a legitimate server	
Malicious Server	Controls a seemingly legitimate server that responds to client requests maliciously, violating protocol rules and compromising security.	Responses to legitimate client requests	

Table 2: Identified Attack Vectors in Network Protocols

These attack vectors serve as the foundation for developing our **Network Attack Model Framework**. In this framework, we emulate different attack scenarios to assess network protocols for potential vulnerabilities. Further details will be provided in the next section. Now, let's explain the Network-Centric Testing framework that underpins our methodology.

(2) Network-Centric Testing Approaches. Network-centric compositional testing (*NCT*) is a specialized approach within model-based testing (*MBT*) that focuses specifically on network protocols. The principle of compositional testing views formal specifications as a system of interrelated components or processes, each with its own input and output. This method allows for examination of the protocol behaviors in their network interactions, instead of relying exclusively on abstract mathematical models. The emphasis on genuine network behavior is what distinguishes the methodology as "network-centric".

NCT Principle. NCT creates a systematic framework for generating formal specifications of Internet protocols and verifying their implementations for conformity to these specifications [14,23]. After generating the formal model code, a test generator produces concrete and randomized test cases. The testers, which use an *SMT solver* to meet the constraints set by the formal protocol requirements, are then deployed to verify the real-world implementation of the protocol. If any protocol requirements are not met, the resulting traces can be examined to find and identify potential errors or vulnerabilities. Let us consider a minimal network protocol designated as `MiniP` to illustrate how *NCT* works.

MiniP. The Minimalist Protocol (`MiniP`) is specified as follows. Each packet must contain exactly two frames. There are three types of frames : `PING`, `PONG`, and `TIMESTAMP` frames. The `PING` and `PONG` frames include a *four-byte string* that corresponds to the word 'ping' or 'pong', respectively. A packet must contain one of these two frames. Additionally, the `TIMESTAMP` frame contains an *eight-byte unsigned integer* that represents the time, in milliseconds, when the packet is transmitted. Every packet includes this frame. The client initiates communication by sending a packet with the `PING` frame and the `TIMESTAMP` frame as its payload. The server is required to reply within *three seconds* with a packet that includes the `PONG` frame followed by the `TIMESTAMP` frame. This process repeats until the client disconnects. To end the connection, the client simply stops sending packets for *more than three seconds*.

MiniP Network-centric Structure. Figure 1 illustrate the design of `MiniP` according to the *NCT* principle. The "Frame" process generates output that is used as input for the "Packet" process. The assumptions regarding the inputs of a process are treated as guarantees for the outputs of other processes. Each element represents a layer of the `MiniP` stack, including the frame layer ⓐ and the packet layer ⓑ. The *shim* component ⓒ handles the network transmission and packet reception. Once a packet is received, the shim component checks the formal specifications associated with the packet. For example, it ensures that a MiniP packet consistently includes two frames in the appropriate sequence. Should any of the criteria not be satisfied, an error is triggered. Frames are likewise handled according to their own formal specifications.

Formal specification. NCT employs the Ivy language [22, 29] for formal specification purposes. Ivy facilitates describing a program's state through first-order

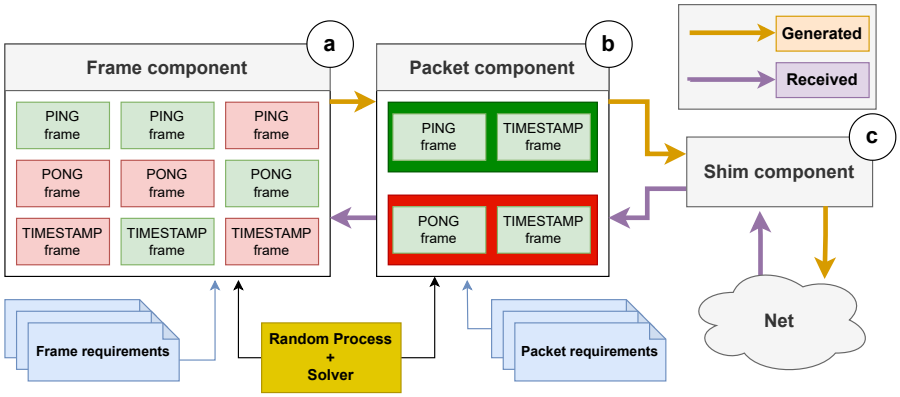

Fig. 1: `MiniP` Network-Centric Testing (*NCT*) structure

logic formulas and incorporates relations (boolean predicates), functions, modules, and type objects as the primary abstractions to model the system's state. For example, the Ivy code snippet 1 defines how the handling of the `PING` frame, which must contain the word 'ping' as a *four-byte string*.

</> 1. Ping frame formal specification code </>

```
before ping_frame.ping.handle {
    require f.data     = ping_data; #data must be 'ping'
    require f.data.end = 4; #length of the data must be 4
}
```

While *NCT* approaches have been effective in identifying protocol compliance issues, it is crucial to extend them to include tests that explore beyond safety properties focus, as outlined in their paper [24].

(3) Network-Centric Testing Limitations. *NCT* and Specification-conforming testing, which focuses on ensuring that implementations strictly adhere to formal specifications, has several inherent limitations:

① *Limited to Specification-Conforming Testing.* Focusing solely on testing that adheres to the specifications restricts the overall extent of coverage. By strictly following the formal specifications, the methodology might overlook potential issues stemming from unexpected or non-specification-conforming inputs and behaviors, which are prevalent in practical scenarios. This aspect is crucial, as real-world networks frequently encounter unforeseen interactions or attacks that are not predicted by a specification. This limitation underscores the necessity to broaden the testing framework to include tests that go beyond the specified boundaries, possibly incorporating formal attack models that mimic adversarial conditions or unusual network behaviors.

② *Single Protocol Testing.* The current approach is designed to test one protocol at a time. This approach limits the framework's applicability to more complex systems where multiple protocols interact, such as in distributed services or

multi-layered web applications. These interactions might introduce unforeseen issues or vulnerabilities. Expanding the methodology to support multi-protocol testing would significantly improve its ability to detect interaction-based vulnerabilities.

③ *Restricted to Point-To-Point communications.* The testing methodology is currently limited to point-to-point communications, which does not fully capture the complexity of modern networked systems that often involve multipoint or broadcast communications. Many real-world network applications require the verification of protocols that manage communications among multiple nodes, such as in mesh networks or distributed systems. Extending the testing framework to handle these more complex communication patterns would significantly enhance its usefulness and applicability.

Putting all together. Addressing these limitations by integrating *formal nonconform (mutation) attack models*, supporting *multi-protocol environments*, and expanding to *multi-node communications* would greatly enhance the robustness and comprehensiveness of protocol testing, ensuring that the protocol performs well not only in controlled environments but also in real-world, dynamic network scenarios.

3 Network Attack Model Framework

Network Attack-centric Compositional Testing (NACT), a method of model-based mutation testing, facilitates the evaluation of network protocols' security and robustness by introducing mutations (minor alterations) to the specifications (*addressing Limitation* ①) and testing these modifications across different use cases (*addressing Limitations* ② & ③). To accomplish this, we suggest a framework for a network attacker model that operates in three phases.

Step1: Defining formal specification mutations. Mutation testing has been successfully applied to network protocols to improve their robustness and security. This approach involves intentionally making small changes (mutations) to the protocol specifications, resulting in mutant versions that might reveal potential weaknesses [9,20,34,39]. Subsequently, the mutants are evaluated against a comprehensive set of test cases to determine the efficacy of existing tests in detecting artificially introduced errors. The main goal is to evaluate the protocol's fault tolerance and to determine potential failure points that may arise in unexpected circumstances.

Table 3 presents various mutation techniques applicable to formal specifications in internet protocol testing. These specific mutations are chosen for their ability to effectively challenge and validate the robustness of protocol implementations.

Furthermore, they specifically target critical protocol behavior aspects that are prone to reveal hidden issues. Other mutations might not be effective in this

scenario, either due to insufficiently significant deviations or the generation of invalid mutants that fail to yield meaningful test results.

Mutation	Description	Problem Investigated	Challenges
Statement Deletion or Addition	Remove or add statements to observe protocol deviations	Functional errors, missing logic, incomplete flows, Protocol deviations, unintended side-effects	Incomplete protocol states, trivial bugs, unreachable or invalid states
Negation Mutation	Negate logical expressions to introduce faults	Logical errors, incorrect conditional handling	Reveal deep logical errors
Value Replacement	Modify variables or constants to test behavior under changes	Edge-cases, faulty variable handling	Maintain meaningful tests
Boundary Value Mutation	Alter values to boundary limits for edge-case testing	Buffer overflows, boundary issues	Select boundaries carefully to avoid meaningless mutations
Control Flow Mutation	Modify control structures to test robustness	Conditional logic failures, Infinite loops, performance bottlenecks	Unreachable states, complex conditions, computational cost
Mutate Data Structures	Change data structure definitions to find vulnerabilities	Memory management, invalid state transitions	Ensure valid structure mutations

Table 3: Formal Specification Mutations in Internet Protocol Testing.

At the protocol layer, these mutations are incorporated into both the frame and packet components. Figure 2 illustrates the integration of mutations within `MiniP`'s architecture. Based on the requirements, modifications are made to define the mutations, which are subsequently embedded in the protocol components, thereby exploring how the implementation handles these deviations from the standard specification (*Limitation* ①).
To test these mutations effectively, we must now implement them. This is where *NCT* becomes relevant.

Step2: Implementing composable attacker specifications. Our approach for implementing mutations extends the conventional formal specification mutation testing framework by introducing mutations directly into the formal specification. This is facilitated by *NCT*. Currently, generating mutants is a manual process due to the difficulty of generalizing mutations across all network protocols while maintaining consistent and meaningful system behavior. To ensure a degree of generality, we provide a template for each attack model: the man-in-the-middle, malicious client, and server (Table 2).

At a protocol level, we show below, using our running protocol `MiniP`, how to implement a malicious character injection (a mutation by addition), find possible security vulnerabilities (vs safety vulnerabilities) in this mutant, and implement a test scenario to test the vulnerability.

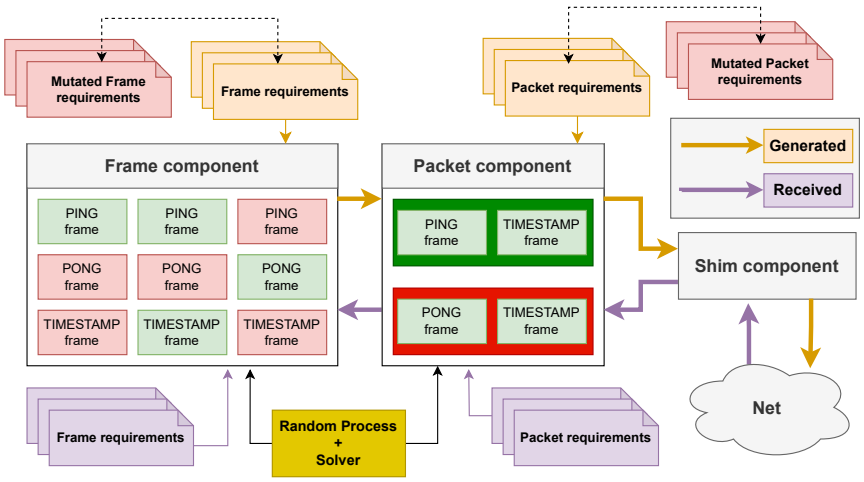

Fig. 2: `MiniP` Formal Specification Mutation

(1) Finding vulnerability with specification mutation. The specific mutation introduced here consists in altering a requirement to allow the inclusion of potentially harmful characters in the packet payload. The following Ivy code snippet 2 represents this mutation showing the modification of the original ping frame handler presented in the code snippet 1 Sect.2.

2. Format String & Buffer Overflow

```
before ping_frame.ping.handle_maliciously {
    require f.data.end < 100000 & exists I. I < f.data.end-8  &
      (f.data.value(I)   = 0x25  & f.data.value(I+1) = 0x78  &
       f.data.value(I+2) = 0x25  & f.data.value(I+3) = 0x6e);
}
```

This mutation is designed to evaluate the `MiniP`'s robustness against format string vulnerabilities, a well-known category of security issues. In addition, we varied the size of the message payload to assess the protocol's handling of unusually large or small messages, a common technique for identifying buffer overflow vulnerabilities. This vulnerability is particularly severe, as it can result in memory corruption, leading to potential system crashes or arbitrary code execution by an attacker. Once a vulnerability is identified using randomized mutation testing, the next step involves creating specific scenario test cases.

(2) Exploiting vulnerability creating a test scenario A scenario test case is designed to explicitly target the identified vulnerabilities and structured to replicate the exact conditions under which the vulnerability was found. The test scenarios are created using attack vectors (Table 2). For example, we can replicate the protocol's behavior during a *Man-in-the-Middle attack* or its reaction to *malicious clients* sending improperly formatted packets under mutations. By incorporating these vectors into mutation testing, we can validate the system's

ability to withstand real-world attacks. These scenario-based tests can be reused in future tests to ensure that the vulnerability remains mitigated in subsequent implementations or protocol versions. One can use *NCT* to craft test scenarios with straightforward assertion directives. The detected vulnerabilities (string format and buffer overflow) can be tested using the following code snippet (3).

</> 3. Malicious Character Injection Specific test </>

```
before ping_frame.ping.handle_maliciously {
    require f.data.end = 50; #Buffer overflow size + format string
    require f.data.value(6) = 0x25 & f.data.value(7) = 0x78;
}
```

Now that we define mutations and test scenarios, we need to test the protocol under test in an appropriate environment.

Step3: Simulating attack scenarios within virtual networks. A crucial stage in developing and validating network protocols involves thorough testing to confirm that protocols operate as expected under various network conditions. Our methodology highlights the significance of testing in settings that closely resemble real-world scenarios. Consequently, we mainly use *virtual networks*.

(1) Using virtual networks. Virtual networks, by leveraging namespaces, enable the creation of isolated network environments that accurately emulate actual deployment settings. This approach facilitates the testing of multipoint communications (*Limitation* ③) and complex topologies, imitating real-world network dynamics. Building upon *NCT*, our framework is compatible with simulators. These simulators provide various benefits, including reproducibility, accurate management of network conditions, and the ability to simulate intricate scenarios such as timing attacks [30]. However, employing simulators necessitates the creation of simulated system calls (syscalls) to emulate the network stack, which can be challenging in proportion to the complexity of the simulation condition. Extending *NCT* to complex network systems, especially in microservices architectures, involves dealing with the added complexity of interacting protocols. Figure 3 illustrates the interaction between our framework and a virtual network. The *NCT* framework integrated with an attack model ⓐ produces model-based testers or attackers ⓑ, which are used within the virtual network consisting of multiple client-tested implementations ⓒ interfaced with a server-tested implementation. Upon completion of the simulation, the network traces are analyzed to investigate vulnerabilities.

(2) Compositional Testing of Networks Services. *NCT* originally focused on testing single protocols through formal specifications and automated testers, ensuring compliance via *SMT solvers* [23]. In a microservice environment, where services may use different protocols (e.g., HTTP over TCP, gRPC over `QUIC`), the challenge is to develop modular formal specifications that capture both individual service behavior and inter-service interactions [7]. To do so, we design the framework with some level of abstraction, illustrated in Figure 4. The *system*

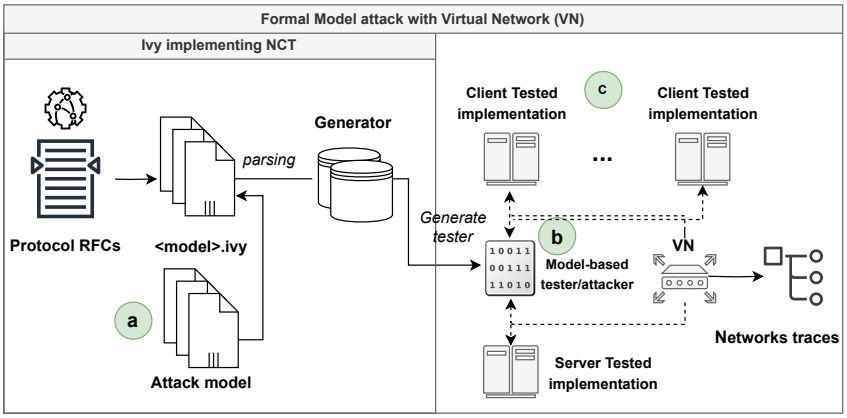

Fig. 3: Virtual networks and *NACT*

component, a virtual component, serves as a master framework that defines base events common across all protocols, ensuring consistent integration and testing (*Limitation* ②). The model can then be expanded with *protocol-specific requirements*, allowing each protocol to be tested both in isolation and as part of the broader system. A *shim* layer manages interactions between different protocols, routing events to the appropriate components and ensuring that protocol-specific requirements are applied while maintaining system integrity.

We implemented a toy example demonstrating a *MitM* attack scenario where `QUIC` packets are reflected towards `MiniP` endpoints ending in buffer overflow. This simple setup provides a valuable scenario into the interaction between different protocols under attack conditions, highlighting potential vulnerabilities in cross-protocol communication environments.

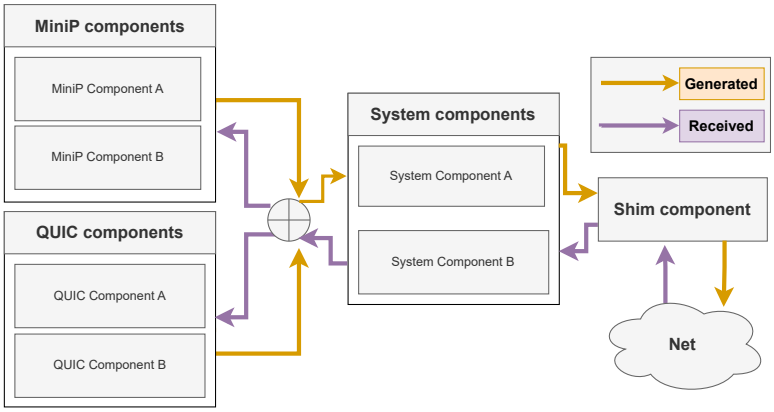

Fig. 4: System Component

Putting all together. The proposed Network Attack-centric Compositional Testing (*NACT*) framework provides a structured and comprehensive approach to enhancing the security and robustness of network protocols. This framework

integrates the mutation of formal specifications, the deployment of attacker specifications that can be composed, and the simulation of attack scenarios in virtual network environments to provide a flexible approach for uncovering hidden vulnerabilities. Our method performs specification-conforming *and* non-conforming testing, managing both individual protocol and multiple protocol interactions in multipoint communication settings.

In the subsequent section, we will demonstrate how this framework is applied to the testing of the real world `QUIC` protocol and its efficacy in uncovering new vulnerabilities.

4 Experiments

This section is organized into two parts. The first part focuses on reproducing known vulnerabilities in various `QUIC` [19] implementations, such as `picoquic`, `quic-go`, `quinn`, `quant`. We apply established attack scenarios to both current and previous versions of these implementations to verify the effectiveness of security patches and identify any regressions. The second part involves the discovery of new vulnerabilities using our Network Attack-centric Compositional Testing (NACT) framework on `lsquic` and `quant` implementation.

(1) Testing recent `QUIC` vulnerabilities. To evaluate vulnerabilities within our framework, it is required to define both the vulnerabilities and the associated attack scenario, given the mutations are defined (Section 3). Table 5 shows the chosen `QUIC` vulnerabilities and the respective attack scenario designed to address them.

Implementation	**Language**	**SLOC**	**Version**
[11] `picoquic` (a)	C	122k	bb6799 (actual)
`picoquic` (b)	C	84k	42c620 (draft 27)
[17] `quic-go` (a)	Go	83k	v0.46.0
`quic-go` (b)	Go	73k	b5ef99a
[31] `quinn` (a)	Rust	33k	0.11.2
`quinn` (b)	Rust	41k	0.10.0 (draft 29)
[25] `quant` (a)	C	18k	dc7721
`quant` (b)	C	18k	bf903d (draft 29)

Table 4: Tested implementations informations

To thoroughly assess the security of `QUIC` implementations in relation to these vulnerabilities, we performed tests on both the latest and older versions of several widely used libraries. Each `QUIC` implementation was examined for known vulnerabilities using its most recent version as well as a previous version to detect any regressions or persistent issues. By testing multiple versions, we

N°	Vulnerability	Description	Known Causes	Occurred Problems	Attack Scenario
(1)	CVE-2022-30591 `QUIC` Slow Loris [1]	DoS via incomplete `QUIC` or HTTP/3 requests	Misparsing of the MTU Discovery service by the *`quic-go`* implementation	CPU consumption and DoS via *Slowloris* attack	Maintain the connection with minimal data transfer, forcing the server to hold resources indefinitely [6]
(2)	CVE-2023-42805 `QUIC` Frame Parsing Exploitation [2]	Incorrect parsing of unknown `QUIC` frames causing erroneous responses	Reception of unknown frames within `QUIC` packets prior to fix	Erroneous server responses	Exploit frame parsing vulnerabilities to manipulate the server response
(3)	CVE-2024-22189 `QUIC` Memory Exhaustion [3]	Memory exhaustion through excessive transmission of `NEW_CONNECTION_ID` frames	Exploitation of congestion control by collapsing the window	Denial of service, memory exhaustion	Flood the server with requests to retire connection IDs, overwhelming the server's resources
(4)	GitHub issues *`picoquic`* Malicious Payload Injection [27]	Malicious `QUIC` frame triggers an infinite loop, causing a DoS	Crafting maliciously `QUIC` frames processed during session epoch 3	Infinite loop, server crash, remote exploitation	Inject crafted payloads to trigger an infinite processing loop
(5)	GitHub issues `quant` Malicious Payload Injection [26]	DoS caused by processing connections with identical connection IDs	Duplicating connection IDs processed by `quant`	Server crash and denial of service	Inject frames to cause server failure and potentially execute malicious payloads
(6)	Non-conformance bug in `QUIC` Implementations (`NEW_TOKEN` frame) [5]	Several `QUIC` implementations (`picoquic` 7f2fbdfb & `lsquic` 3.2.0) fail to handle invalid packets properly	Protocol specification non-conformance	Invalid error codes and server instability	Replay malformed `NEW_TOKEN` frames to reproduce server errors and non-conformance

Table 5: `QUIC` selected vulnerabilities and our defined attack scenarios

ensure that vulnerabilities identified in earlier versions are indeed present and can be reliably reproduced, while also verifying that these vulnerabilities have been effectively fixed in the most recent versions, preventing their reintroduction.

To guarantee the dependability and consistency of our results, each test was executed three times across the different `QUIC` implementations. Due to the variability inherent in some tests, this approach allowed us to control for any inconsistencies in the data and better evaluate the robustness of each implementation against the identified vulnerabilities. Table 6 displays the summary of these tests' outcomes. For each test, we allocated a time budget of 150 seconds per test [13].

Implem. / Attacks	`picoquic` (a)	`picoquic` (b)	`quic-go` (a)	`quic-go` (b)	`quinn` (a)	`quinn` (b)	`quant` (a)	`quant` (b)
(1)	✗	✗	✗	∼	✗	✗	✗	✗
(2)	✗	✗	✗	✗	✗	∼	✓	✓
(3)	✗	✗	✗	∼	✗	✗	✗	✓
(4)	✗	✓	✗	✗	✗	✓	✗	✓
(5)	✗	✗	✗	✗	✗	✗	✗	✓
(6)	✗	✓	✗	✗	✗	✓	✗	✓
Results	*safe*	*unsafe*	*safe*	*unsafe*	*safe*	*unsafe*	*unsafe*	*unsafe*

Table 6: Reproduction of Vulnerability Testing Results for Various QUIC Implementations. QUIC Implementations versions: (a) *actual* and (b) *old*. Each symbol represents the outcome of the tests, running three times: ✓ indicates that the three trials failed (vulnerable), ✗ indicates that the three trials passed (safe), and ∼ indicates mixed results (some trials passed and some failed). The attacks numbers (i), 1 to 6, are described in Table 5. Grey cells represent the implementation(s) originally detected as vulnerable to the attack scenario. An implementation is considered *safe* if it passes all tests and *unsafe* otherwise.

The experiments carried out provided valuable information on the behavior and vulnerabilities of various `QUIC` implementations. As shown in Table 6, our testing revealed that while some implementations demonstrated resilience against specific attack scenarios, others exposed vulnerabilities that could lead to potential security risks. Notably, our findings highlighted the effectiveness of newer versions in addressing previously identified issues, showcasing the ongoing efforts to enhance the security and robustness of `QUIC` implementations.

The results indicate that while certain implementations, such as `quic-go` (a), showed no significant issues, others, like `quinn` (b) and `quant` (b), revealed persistent vulnerabilities or improper handling of edge cases, as detailed in the various test outcome.

One of the key observations was the effectiveness of newer versions in addressing previously identified vulnerabilities. For example, in the case of `picoquic`, the vulnerabilities observed in version (b), which included issues concerning the

frame parsing process and RFC compliance, were successfully mitigated in the newer version (a), demonstrating a clear improvement in security.

However, the experiments also highlighted some persistent challenges. Certain implementations, such as `quinn` (b), showed issues in handling edge cases, such as repeated sending of `CONNECTION_CLOSE` frames (> 10) followed by Stateless Reset packets, indicating a need for further refinement. Similarly, `quant` (b) frequently experienced crashes or improper connection closures when dealing with oversized tokens, illustrating the difficulties in achieving robust `QUIC` implementations across different scenarios.

The experiments faced some difficulties, particularly in consistently triggering specific vulnerabilities like the Slowloris attack on `quic-go` (b). This challenge underscores the complexity of certain attack vectors and the ambiguous nature of some CVE designations, which can make reproducibility and consistent testing outcomes difficult.

In conclusion, these results underscore the significance of comprehensive and rigorous testing frameworks for the assessment of protocol security. The varying degrees of resilience and vulnerability observed across different versions and implementations underscore the necessity for continuous vigilance, rigorous testing, and the refinement of security measures. By addressing these challenges, the security and robustness of `QUIC` and similar network protocols can be significantly enhanced, ensuring their resilience against evolving threats.

(2) Discovering new network protocol vulnerabilities. In the course of our investigation into `QUIC` protocol implementations, we identified a number of vulnerabilities that could potentially be exploited under certain conditions. This section presents our findings, with a particular focus on scenarios involving MitM and client-initiated attacks.

(2.1) Version Negotiation Abuse - Denial of Service The LiteSpeed `QUIC` (`lsquic`) Library is a fast, flexible, and production-ready open-source implementation of `QUIC` and HTTP/3 for servers and clients. It has been used in LiteSpeed products since 2017 and supports multiple `QUIC` versions. With around 129k SLOC, `lsquic` is significantly larger compared to other implementations. A vulnerability was identified with regard to the version negotiation process. In particular, when `lsquic` initiates a handshake using version 0xff000022 (draft-34) and receives a version negotiation packet responding with 0xff00001d (draft-29), the implementation erroneously applies an incorrect encryption key to subsequent packets. This results in checksum failures, which could potentially lead to a denial of service (*DoS*) condition.

It is noteworthy that this issue does not manifest when transitioning between certain other versions, such as from 0xff00001b (draft-27) to 0xff00001d (draft-29). However, the vulnerability is consistently manifested when moving from version 0xff000022 (draft-34) to 0xff00001b (draft-27). Testing the reverse transition (from 0xff00001d to 0xff000022) is challenging with the current `lsquic` setup, as it defaults to selecting the highest available version.

The root cause of this vulnerability is not related to the key generation process, as each version employs a distinct salt for deriving encryption keys. Rather, the issue arises from the inappropriate application of these keys following version negotiation. This vulnerability presents a potential vector for a MitM attack. While initial packets are encrypted, they do not fully protect against MitM attacks. Furthermore, version negotiation packets, as defined in RFC9000, are entirely unprotected. Consequently, an attacker could intercept and manipulate the version negotiation packet, either by altering the selected version or crafting a fraudulent packet. This could potentially lead to a denial-of-service (DoS) attack.

This vulnerability appears to be specific to the `lsquic` implementation and is not necessarily indicative of a flaw in RFC9000 itself. However, the RFC's relatively vague guidelines on the version negotiation process could contribute to implementation inconsistencies, leading to potential vulnerabilities like the one identified, where an off-path attacker could exploit the protocol's flexibility to deplete server resources.

(2.2) Malformed Frame Encodings - Memory exhaustion attacks Exploits In the course of our preliminary experiments with a modified specification, we identified a significant error in the implementation of `QUIC`'s `quant` (a) functionality. In particular, when a specific type of frame encoding was employed, the `quant` process entered an infinite loop during the connection close procedure. This issue was particularly evident when the client transmitted a `NEW_TOKEN` frame, which is not permitted by the RFC, in conjunction with a parsing error in the HTTP request contained within the `STREAM` frame. These errors resulted in an infinite loop, as the frame encodings differed slightly from the anticipated format but remained within the acceptable range, prompting the `quant` algorithm to classify them as valid input. Consequently, the connection close state was not correctly exited, leading the server to continue consuming resources and potentially resulting in a system crash.

A subsequent test with a different mutation of the specification yielded comparable results, though through a distinct mechanism. In this scenario, the altered frame encoding modified certain fields within the `NEW_CONNECTION_ID` frame header in a way not anticipated by the original specification. The parsing error in the HTTP request exacerbated the situation, leading to repeated, unsuccessful attempts by the `"quant"` process to close the connection. The unexpected packet structure caused the connection closure process to enter an indefinite loop.

In both cases, the infinite loop was triggered when the server initially sent a `CONNECTION_CLOSE` frame with a `PROTOCOL_VIOLATION` error code twice, followed by an endless sequence of HTTP 505 errors encapsulated within `FLOW_CONTROL_ERROR` frames. In the context of the mutated specification and introduced parsing errors, this combination of frames resulted in the persistent failure of the connection termination process, which ultimately caused the server to become unresponsive.

Notably, this issue does not appear in older versions of `quant` (b), suggesting that it has been introduced in more recent updates. This highlights the importance of continuous testing across different versions to identify and address newly introduced vulnerabilities.

5 Conclusion & Future works

Robust network protocol testing is crucial in an interconnected digital world due to evolving cyber threats. Our research explores attack vectors and develops a comprehensive, mutation-based testing framework to identify vulnerabilities. We addressed several attacks by constructing the *Network Attack-centric Compositional Testing (NACT)* framework. Mutations generate adversarial conditions by modifying protocol specifications, exposing hidden vulnerabilities that traditional tests might miss, especially within formal specification testing (*NCT*).

While *NCT* has laid the groundwork for testing protocol compliance, it inherits some limitations. One inherited challenge is its focus on specification conformance, which leaves out adversarial behavior and multi-protocol interactions. Another challenge is the point-to-point communication limitation missing the complexity of modern distributed systems. We addressed these limitations by incorporating non-conformant mutation-based attack models, supporting multi-protocol environments, and extending the framework to multi-node communications using virtual networks. These enhancements enable the framework to perform more comprehensive security testing across diverse network environments.

Our experimental results with `QUIC` demonstrate the efficacy of our framework. We developed consistent attack scenarios to validate and discover vulnerabilities, notably in version negotiation and frame encoding errors, leading to *denial of service (DoS)* or *infinite loop conditions*. These findings underscore the need for robust security testing frameworks for both known and emerging vulnerabilities in protocols such as `QUIC`.

Looking ahead, our research aims to enhance the *NACT* framework by incorporating the *Network Simulator-centric Compositional Testing* (*NSCT*) methodology [30]. NSCT has built upon NCT to verify time-dependent network properties, ensuring reproducibility of experiments via simulators. This approach will aid in replicating attacks and allow the modeling of timing attacks.

We also intend to conduct extensive empirical testing on real network systems associated with the AMC3 project. [4].

Acknoledgements We would like to thank the belgium's "*Defence-related Research Action*" (DEFRA) and the "*Automated Methodology for Common Criteria Certification*" project (AMC3).

References

1. NVD - CVE-2022-30591 (2022), https://nvd.nist.gov/vuln/detail/CVE-2022-30591
2. NVD - CVE-2023-42805 (2023), https://nvd.nist.gov/vuln/detail/CVE-2023-42805
3. NVD - CVE-2024-22189 (2024), https://nvd.nist.gov/vuln/detail/CVE-2024-22189
4. AMC3: What is AMC3 ? (08 2024), https://www.amc3.be/
5. Asadian, H., Fiterau-Brostean, P., Jonsson, B., Sagonas, K.: Monitor-based testing of network protocol implementations using symbolic execution. In: Proceedings of the 19th International Conference on Availability, Reliability and Security. ARES '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3664476.3664521, https://doi.org/10.1145/3664476.3664521
6. Balaji, A.S., Anil Kumar, V., Amritha, P.P., Sethumadhavan, M.: QUICLORIS: A slow denial-of-service attack on the quic protocol. In: Dubey, A.K., Sugumaran, V., Chong, P.H.J. (eds.) Advanced IoT Sensors, Networks and Systems. pp. 85–94. Springer Nature Singapore, Singapore (2023)
7. Bestavros, A., Kfoury, A., Lapets, A., Ocean, M.: Safe compositional network sketches: Tool & use cases. Conference/Journal Name pp. 234–245 (2009)
8. Bhanpurawala, A., El-Fakih, K., Zualkernan, I.: A formal assisted approach for modeling and testing security attacks in IoT edge devices (10 2022). https://doi.org/10.48550/arXiv.2210.05623
9. Budd, T.A., Gopal, A.S.: Program testing by specification mutation. Computer Languages **10**(1), 63–73 (1985). https://doi.org/https://doi.org/10.1016/0096-0551(85)90011-6, https://www.sciencedirect.com/science/article/pii/0096055185900116
10. Chatzoglou, E., Kouliaridis, V., Karopoulos, G., Kambourakis, G.: Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study. International Journal of Information Security **22** (12 2022). https://doi.org/10.1007/s10207-022-00630-6
11. Christian Huitema: picoquic, https://github.com/private-octopus/picoquic, 4f11445
12. Crochet, C.: PANTHER: Protocol formal analysis and formal network threat evaluation resources - nordsec commit (08 2024), https://github.com/ElNiak/PANTHER/tree/4804a5afba6bc81241c51f6ca71f99e560859fca
13. Crochet, C.: PANTHER: Protocol formal analysis and formal network threat evaluation resources - results (08 2024), https://github.com/ElNiak/PANTHER-Ivy/tree/cf8b80f7bfda33cdd6d285c0110bb3198de28df2/protocol-testing/apt/test/nordsec
14. Crochet, C., Rousseaux, T., Piraux, M., Sambon, J.F., Legay, A.: Verifying QUIC implementations using ivy. Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC (2021). https://doi.org/10.1145/3488660.3493803
15. Everson, D., Cheng, L., Zhang, Z.: Log4shell: Redefining the web attack surface. In: Proc. Workshop Meas., Attacks, Defenses Web (MADWeb). pp. 1–8 (2022)
16. Farooq, F., Nadeem, A.: A fault based approach to test case prioritization. In: 2017 International Conference on Frontiers of Information Technology (FIT). pp. 52–57 (2017). https://doi.org/10.1109/FIT.2017.00017

17. quic go: Github - quic-go/quic-go: A QUIC implementation in pure go (08 2024), https://github.com/quic-go/quic-go
18. Hoque, N., Bhuyan, M.H., Baishya, R., Bhattacharyya, D., Kalita, J.: Network attacks: Taxonomy, tools and systems. Journal of Network and Computer Applications **40**, 307–324 (2014). https://doi.org/https://doi.org/10.1016/j.jnca.2013.08.001, https://www.sciencedirect.com/science/article/pii/S1084804513001756
19. Iyengar, J., Thomson, M.: Rfc 9000, https://www.rfc-editor.org/rfc/rfc9000
20. Li, J.h., Dai, G.x., Li, H.h.: Mutation analysis for testing finite state machines. In: 2009 Second International Symposium on Electronic Commerce and Security. vol. 1, pp. 620–624 (2009). https://doi.org/10.1109/ISECS.2009.158
21. Lu, Y., Xu, L.D.: Internet of things (iot) cybersecurity research: A review of current research topics. IEEE Internet of Things Journal **6**(2), 2103–2115 (2019). https://doi.org/10.1109/JIOT.2018.2869847
22. McMillan, K.L., Padon, O.: Ivy: A multi-modal verification tool for distributed algorithms. Computer Aided Verification p. 190–202 (2020). https://doi.org/10.1007/978-3-030-53291-8_12
23. McMillan, K.L., Zuck, L.D.: Compositional testing of internet protocols. 2019 IEEE Cybersecurity Development (SecDev) (2019). https://doi.org/10.1109/secdev.2019.00031
24. McMillan, K.L., Zuck, L.D.: Formal specification and testing of QUIC. Proceedings of the ACM Special Interest Group on Data Communication (2019). https://doi.org/10.1145/3341302.3342087
25. NTAP: Github - NTAP/quant: QUIC implementation for POSIX and IoT platforms (2016), https://github.com/NTAP/quant
26. NTAP: Dos attack: Server crashes when processing new connections ids that have the same cid. issue 68. NTAP/quant (08 2020), https://github.com/NTAP/quant/issues/68
27. private octopus: Denial of service vulnerability (infinite loop) while parsing malicious QUIC frame. issue 969. private-octopus/picoquic (07 2020), https://github.com/private-octopus/picoquic/issues/969
28. Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: International Conference on the Unified Modeling Language. pp. 416–429. Springer (1999)
29. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety verification by interactive generalization. ACM SIGPLAN Notices **51**(6), 614–630 (2016). https://doi.org/10.1145/2980983.2908118
30. Rousseaux, T., Crochet, C., Aoga, J., Legay, A.: Network simulator-centric compositional testing. In: Castiglioni, V., Francalanza, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. pp. 177–196. Springer Nature Switzerland, Cham (2024)
31. quinn rs: Github - quinn-rs/quinn: Async-friendly QUIC implementation in rust (05 2024), https://github.com/quinn-rs/quinn
32. Rutherford, M.J., Carzaniga, A., Wolf, A.L.: Simulation-based test adequacy criteria for distributed systems. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. p. 231–241. SIGSOFT '06/FSE-14, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1181775.1181804, https://doi.org/10.1145/1181775.1181804
33. Shahriar, H., Zulkernine, M.: Mutation-based testing of format string bugs. In: 2008 11th IEEE High Assurance Systems Engineering Symposium. pp. 229–238 (2008). https://doi.org/10.1109/HASE.2008.8

34. Sidhu, D., Leung, T.K.: Fault coverage of protocol test methods. In: IEEE INFOCOM '88,Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies. Networks: Evolution or Revolution? pp. 80–85 (1988). https://doi.org/10.1109/INFCOM.1988.12901
35. Sui, A.F., Tang, W., Hu, J.J., Li, M.Z.: An effective fuzz input generation method for protocol testing. In: 2011 IEEE 13th International Conference on Communication Technology. pp. 728–731 (2011). https://doi.org/10.1109/ICCT.2011.6157972
36. Tang, W., Sui, A.F., Schmid, W.: A model guided security vulnerability discovery approach for network protocol implementation. In: 2011 IEEE 13th International Conference on Communication Technology. pp. 675–680 (2011). https://doi.org/10.1109/ICCT.2011.6157962
37. Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S.: Pixelvault: Using gpus for securing cryptographic operations. Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (2014), https://api.semanticscholar.org/CorpusID:1457745
38. Wen, S., Meng, Q., Feng, C., Tang, C.: Protocol vulnerability detection based on network traffic analysis and binary reverse engineering. Plos One **12**, e0186188 (2017). https://doi.org/10.1371/journal.pone.0186188
39. Woodward, M.: OBJTEST: an experimental testing tool for algebraic specifications. In: IEE Colloquium on Automating Formal Methods for Computer Assisted Prototying,. pp. 2 pp.– (1992)
40. zaib, R.: Zero-day vulnerabilities: unveiling the threat landscape in network security. MJCS pp. 57–64 (2022). https://doi.org/10.58496/mjcs/2022/007
41. Zarrad, A., Alsmadi, I., Yassine, A.: Mutation testing framework for ad-hoc networks protocols. In: 2020 IEEE Wireless Communications and Networking Conference (WCNC). pp. 1–8 (2020). https://doi.org/10.1109/WCNC45663.2020.9120695