PANDA: An Attack Synthesis Tool for Distributed Protocols*

[1]

Abstract

Distributed protocols are the lynchpin of the modern internet, underpinning every internet service. This has in turn motivated a massive amount of research ensuring the security, reliability, and performance of distributed protocols. A wide-ranging assumption in these works is assuming distributed protocols operate over *faulty* or *attacker-controlled* channels, where messages can be arbitrarily inserted, dropped, replayed, or reordered. Formal methods work formally verifying distributed protocols typically defines their own notion of imperfect or malicious channels, then constructively proves their protocol is correct with respect to it. In this work we take a fundamentally different approach: we develop a rigorous methodology for automatically discovering *attack traces* on distributed protocols with respect to imperfect channels, and we introduce PANDA, a highly generalizable tool for synthesizing attacks on distributed protocols that implements our methodology. PANDA provides sound, complete analysis, synthesizing attacks on arbitrary linear temporal logic (LTL) protocol specifications or proving the absence of such through an exhaustive state-space search. We demonstrate the applicability of PANDA by employing to study TCP, SCTP, and Raft.

1 Introduction

Distributed protocols are the lynchpin of the modern internet, representing the fundamental communication and coordination backbone of all modern services over the internet. The vast importance of distributed protocols has motivated ample research in ensuring their security, reliability, and performance. One popular approach in prior literature has been the employment of *formal methods*, the use of mathematically rigorous techniques, to analyze and verify distributed protocols. Formal methods has been applied to verify Raft [28, 39, 40], Paxos [14, 17, 31], PBFT [17, 32], and countless other distributed protocols [1, 6, 26, 33]. A common assumption among all the prior formal methods work is assuming distributed protocols operate over *imperfect* or *attacker-controlled* communication channels, where messages can be arbitrarily dropped, replayed, or reordered, then proving the targeted protocol maintains properties of interest with respect to them.

In this work, we take a fundamentally different approach to studying distributed protocols under imperfect channels. While previous work generally assumes one specific channel configuration and primarily seeks to manually construct proofs with respect to it, we study protocols under various channel configurations and seek to automatically discover interesting attack traces, counterexamples, and guarantees.

We introduce a general methodology for automatically discovering dropping, replay, reordering attacks traces on distributed protocols, and we introduce PANDA, a highly generalizable tool for synthesizing attacks on distributed protocols that implements our methodology and the drop, replay, and reordering attacker models. PANDA targets the communication channels between the protocol endpoints, and synthesizes attacks to violate general linear temporal logic (LTL) specifications. PANDA is designed to either synthesize an attack, or prove the absence of such via an exhaustive state-space search. PANDA is sound and complete, meaning if there exists an attack PANDA will find it, and PANDA will never have false positives. PANDA is also designed to be easy to use once the protocol model is constructed: all a user must do is select the victim channels, select the desired attacker models, and invoke PANDA.

In this work we take an approach rooted in *formal methods* and *automated reasoning* to construct PANDA. In particular, we employ *model checking*, a sub-discipline of formal methods, to decidably and automatically find attacks in protocols or prove the absence of such.

We summarize our contributions:

- We present generalizable gadgets representing attackers capable of dropping, replaying, and reordering messages

[1] PANDA is an anonymized name for double-blind submission.

on communication channels of distributed protocols.

- We present PANDA, a tool for synthesizing attacks against distributed communication protocols. PANDA supports four general attacker models: an attacker that can drop, replay, reorder, or insert messages on a channel.
- We provide an overview of PANDA and demonstrate its usage by walking through applying it to the ABP protocol
- We present three case studies for three well-known protocols, TCP, Raft, and SCTP, illustrating the usefulness of PANDA.

We release our code and our models as open source at `https://zenodo.org/records/14968780`.

2 Attacker Gadgets

In this section, we introduce our methodology for the automatic synthesis of drop, replay, and reorder attacks on distributed protocols.

We design our gadgets in *generality*; that is, they can be applied to arbitrary communication channels, agnostic to what is actually transmitted. In this section we describe the design for our *drop*, *replay*, and *reorder* gadgets. Fundamentally, these gadgets are designed to be analyzed in respect to composition with the other processes, as described in the gadgetry-based mathematical attack synthesis framework in [36]. Therefore, these gadgets are best realized within program models such as Kripke Structures or I/O automata. Note, these gadgets have been mechanized within SPIN, as further described in 3.

Drop Attacker Gadget Design. The most simple attacker gadget we define is the *drop* gadget, which drops a select number of messages from the victim channel. The user specifies a "drop limit" value that limits the number of packets the attacker can drop from the channel. Note, a higher drop limit will increase the search space of possible attacks, thereby increasing execution time. The dropper attacker model gadget PANDA synthesizes works as follows. The gadget will nondeterministically choose to observe a message on a channel. Then, if the drop limit variable is not zero, it will consume the message. The state diagram of the drop gadget is shown in Figure 1.

Replay Attacker Gadget Design. The next attacker gadget we define is the *replay* gadget, which copies and replays messages back onto a channel. Similarly to the drop limit for the dropping attacker model, the user can specify a "replay limit" that caps the number of observed messages the attacker can replay back onto the specified Channel. The replay attacker gadget works as follows. The gadget has two states, CONSUME and REPLAY. The gadget starts in the CONSUME state and nondeterministically reads (but not consumes) messages

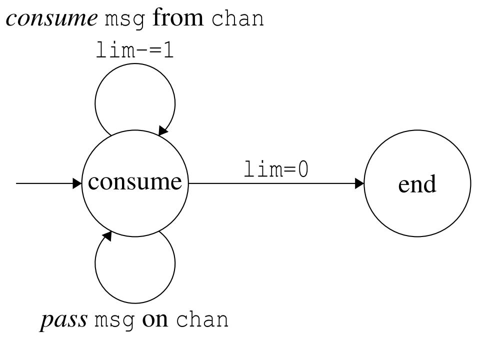

Figure 1: high-level state machine diagram of the *drop* attacker gadget, attacking a channel `chan`. A natural number `lim` is pre-defined.

on the target channel, sending them into a local storage buffer. Once the gadget read the number of messages on the channel equivalent to the defined replay limit, its state changes to REPLAY. In the REPLAY state, the gadget nondeterministically selects messages from its storage buffer to replay onto the channel until out of messages. An example is shown in Figure 2.

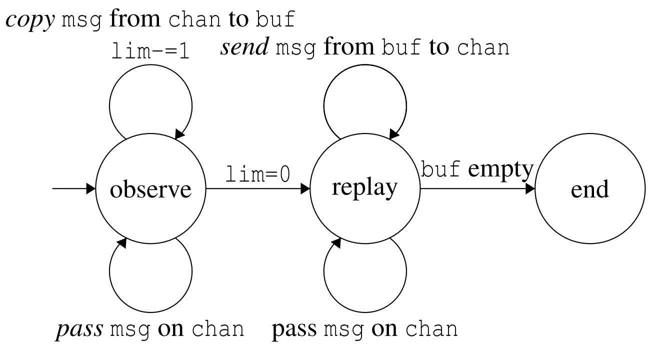

Figure 2: High-level state machine diagram of the *replay* attacker gadget, attacking a channel `chan`. A natural number `lim` is pre-defined. `buf` is a simple FIFO buffer of length `lim`.

Reorder Attacker Gadget Design. The final attacker gadget we define is the *reorder* gadget, which supports reordering messages on a channel. Like the drop and replay attacker model gadgets, the user can specify a "reordering limit" that caps the number of messages that can be reordered by the attacker on the specified channel. The gadget has three states: INIT, CONSUME, and REPLAY. The gadget begins in the INIT state, where it arbitrarily chooses a message to start consuming by transitioning to the CONSUME state. When in the CONSUME state, the gadget consumes all messages that appear on the channel, filling up a local buffer, until hitting the defined reordering limit. Once this limit is hit, the gadget transitions into the REPLAY state. In the REPLAY state, the gadget nondeterministically selects messages from its storage buffer to replay onto the channel until out of messages. An example is shown in Figure 3.

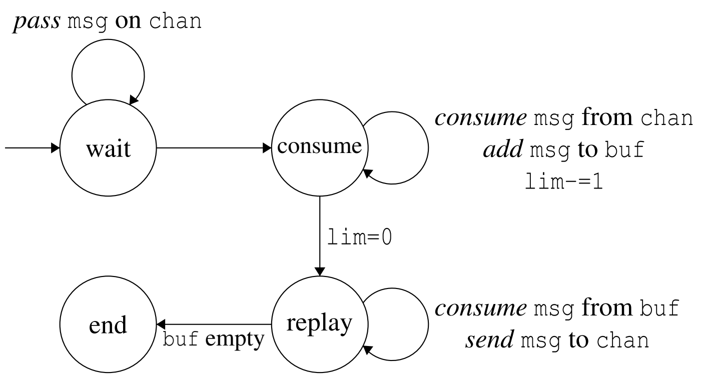

Figure 3: High-level state machine diagram of the *reorder* attacker gadget, attacking a channel chan. A natural number lim is pre-defined. buf is a simple FIFO buffer of length lim.

3 Panda Architecture

In this section we discuss the details behind the design, formal guarantees, implementation, and usage of Panda.

3.1 High-level design

Panda is designed to synthesize attacks with respect to *imperfect* channels. That is, Panda is designed to synthesize attacks that involve replaying, dropping or reordering messages on one or more communication channels relied upon by the victim protocol.

Panda is designed to attack user-specified communication channels in state machine-based formal models of distributed protocols. To use Panda, the user inputs a formal model of a distributed protocol in the Promela language, the communication channel(s) in the protocol model they wish to attack, the desired attacker model, and a formalized correctness property for the protocol model. The protocol model should satisfy the correctness property in absence of Panda.

Once Panda is invoked, it will modify the user-inputted Promela model such that it integrates the desired attacker model. Then, Panda passes the updated Promela model to the model checker which performs the exhaustive search for an attack, returning a trace if such an attack is found.

A high-level visual overview of the Panda pipeline is given in Figure 4.

3.2 Panda Implementation

The SPIN Model Checker. We choose to implement Panda on top of the Spin, a popular and robust model checker for reasoning about distributed and concurrent systems. Spin has existed for over 40 years, and has been applied to dozens of real systems including the Mars Rover [18], Path-Star

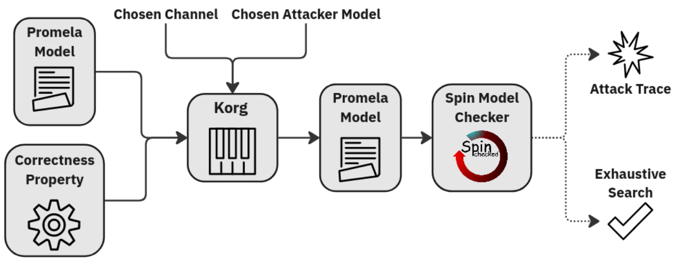

Figure 4: A high-level overview of the Panda workflow

Access server [19], and an avionics operating system [35]. Additionally, Spin has spawned a dedicated formal methods symposium, currently in its 32nd year[2], and earned the 2002 ACM Software System award. Alternatively, the Panda could easily be built on top of other automated reasoning tools besides Spin, including TLA+ using the methodology described in [37], Dafny using the formulation of I/O automata described in [21], and mCRL2 using its process algebra-based modeling framework [9]. We choose Spin over these options due to its historical popularity and robustness.

Intuitively, models written in Promela, the modeling language of Spin, are communicating state machines whose messages are passed over defined *channels*. Channels in Promela can either be unbuffered *synchronous* channels, or buffered *asynchronous* channels. Panda generates attacks *with respect* to these defined channels.

```
// channel of buffer size 0
chan msg_channel = [0] of { int }

active proctype Peer1() {
    msg_channel ! 1;
}

active proctype Peer2() {
    int received_msg;
    msg_channel ? received_msg;
}
```

Listing 1: Example Promela model of peers communicating over a channel. ! indicates sending a message onto a channel, ? indicates receiving a message from a channel.

Panda is designed to parse user-chosen channels and generate gadgets for sending, receiving, and manipulating messages on them. Panda has built-in gadgets that are designed to emulate various real-world attacker models. Once one or multiple gadgets are generated, Panda invokes Spin to check if a given property of interest remains satisfied in the presence of the attacker gadgets.

Attacker Gadget Implementations. Panda supports synthesizing three general attackers, implementing the gadgets

[2]https://spin-web.github.io/SPIN2025/

as described in section 2: attackers that can drop, replay, or reorder messages on a channel. We now discuss the various details that went into the implementations of the various attacker gadgets within SPIN.

Drop Attacker Gadget Implementation. The most simple model PANDA implements is an attacker that can *drop* messages from a channel. The gadget works as follows. A pre-defined limit value `lim` is set, and the attacker begins in the `MAIN` state. Whenever a message is observed on a channel, if `lim == 0` the gadget progresses to the `BREAK` state. Otherwise, the observed message is consumed, `lim` is decremented, and the gadget returns to the `MAIN` state. An example *drop* gadget automatically synthesized with PANDA against the channel `cn` is shown in figure 5.

```
chan cn = [8] of { int, int, int };

active proctype attacker_drop() {
int b_0, b_1, b_2;
byte lim = 3; // drop limit
MAIN:
  do
  :: cn ? [b_0, b_1, b_2] -> atomic {
    if
    :: lim == 0 -> goto BREAK;
    :: else ->
       cn ? b_0, b_1, b_2; // consume
           message on the channel
       lim = lim - 1; goto MAIN;
    fi }
  od
BREAK: }
```

Figure 5: Example dropping attacker model gadget with drop limit of 3, targetting channel "cn"

Replay Attacker Gadget Implementation. The next model PANDA implements is an attacker that can *replay* messages on a channel. The gadget works as follows. The attacker model gadget comes with a pre-defined message limit value `lim`, which defines the length of the gadget storage FIFO buffer `gadget_mem`. Then, the gadget enters the `CONSUME` state and nondeterministically chooses messages on the channel `cn` to copy into `gadget_mem`. Once `lim=0`, the gadget transitions into the state `REPLAY`, where items are randomly selected from `gadget_mem` to be replayed back onto the channel. Note, because `gadget_mem` is a FIFO buffer, we must rotate messages within the channel in order to randomly select a value from it. Additionally, messages can also be nondeterministically removed from `gadget_mem`, as all messages do not necessarily need to be replayed. Once `gadget_mem` is empty, the gadget transitions to the `BREAK` state. An example gadget automatically synthesized with PANDA against the channel `cn` is shown in figure 6.

```
chan cn = [8] of { int, int, int };

// local memory for the gadget
chan gadget_mem = [3] of { int, int, int };

active proctype attacker_replay() {
int b_0, b_1, b_2; int lim = 3;
CONSUME:
  do
  // read messages until the limit is passed
  :: cn ? [b_0, b_1, b_2] -> atomic {
   cn ? <b_0, b_1, b_2> -> gadget_mem !
       b_0, b_1, b_2;
    lim--;
    if
    :: lim == 0 -> goto REPLAY;
    :: lim != 0 -> goto CONSUME;
    fi }
  od
REPLAY:
  do
  :: atomic {
    // nondeterministically select a random
        value from the storage buffer
    int am;
    select(am : 0 .. len(gadget_mem)-1);
    do
    :: am != 0 ->
      am = am-1;
      gadget_mem ? b_0, b_1, b_2 ->
          gadget_mem ! b_0, b_1, b_2;
    :: am == 0 ->
      gadget_mem ? b_0, b_1, b_2 -> cn !
          b_0, b_1, b_2;
      break;
    od }
  // doesn't need to replay all stored msgs
  :: atomic {gadget_mem ? b_0, b_1, b_2; }
  // once mem has no more messages, we're
      done
  :: empty(gadget_mem) -> goto BREAK;
  od
BREAK: }
```

Figure 6: Example replay attacker model gadget with the selected replay limit as 3, targetting channel "cn"

Reorder Attacker Gadget Implementation. The final and most complex gadget PANDA implements is an attacker that can *reorder* messages on a channel. The gadget works as follows. The attacker model gadget comes with a pre-defined message limit value `lim`, which defines the length of the gadget storage FIFO buffer `gadget_mem`. The gadget also has the highest *execution priority* in the system, which ensures the gadget can always reorder messages on the victim channel without other processes interfering. The gadget first enters the INIT state — in this state, the gadget non-deterministically chooses to pass messages on the victim channel, or transition to the CONSUME state. In the CONSUME state, the gadget consumes each message it sees and stores them in gadget_mem, a FIFO buffer. Upon consuming each message from the victim channel, lim is decremented. Once lim=0, the gadget transitions to REPLAY. Then, messages are randomly selected from gadget_mem to be replayed back onto the channel. Note, because gadget_mem is a FIFO buffer, we must rotate messages within the channel in order to randomly select a value from it. Additionally, messages can also be nondeterministically removed from gadget_mem, as all messages do not necessarily need to be replayed. Once gadget_mem is empty, the gadget transitions to the BREAK state. An example gadget automatically synthesized with PANDA against the channel cn is shown in figure 7.

Preventing PANDA Livelocks. In general, there are two types of LTL properties: safety, and liveness. Informally, safety properties state "a bad thing never happens," and liveness properties state "a good thing always happens." Therefore, safety properties can be violated by finite traces, while liveness properties require infinite traces to be violated. When evaluating a PANDA attacker model gadget against a PROMELA model and a liveness property, it is crucial to ensure the gadget has no cyclic behavior. If a PANDA gadget has cyclic behavior in any way, it will trivially violate the liveness property and produce a garbage attack trace. To prevent this, we make the following considerations.

First, we design our PANDA gadgets such that they never arbitrarily send and consume messages to a single channel. Second, we allow PANDA gadgets, which are always processing messages on channels, to arbitrarily "skip" messages on a channel if need be. We implement message skipping by arbitrarily stopping and waiting after observing a message on a channel; once the channel is observed changing lengths, the message is considered skipped and future messages can be consumed.

Passing Messages On Channels. In order to arbitrarily pass messages on channels in SPIN, required functionality for all three gadgets we present, we exploit the finite channel length assumption. Our SPIN gadget is shown below.

```
inline wait_until_pass(chan ch){
  cn ? [b_0, b_1, b_2] -> {
    // wait for chan to change
        lengths. then, exit the loop
    blocker = len(cn);
    do
    :: blocker != len(cn) -> break;
    od }
}
```

That is, when we enter wait_until_pass, we track the length of the channel cn and wait until len(cn) changes. This allows us to arbitrarily pass messages on a given channel cn without reasoning directly about the message.

```
chan cn = [8] of { int, int, int };

// local memory for the gadget
chan gadget_mem = [3] of { int, int, int };

active proctype attacker_reordering()
    priority 255 {
byte b_0, b_1, b_2, blocker; int lim = 3;
INIT:
do
  :: wait_until_pass(ch);
  :: goto CONSUME;
od
CONSUME:
do
  // consume messages with high priority
  :: c ? [b_0] -> atomic {
    c ? b_0 -> gadget_mem ! b_0; lim--;
    if
    :: lim == 0 -> goto REPLAY;
    :: lim != 0 -> goto CONSUME;
    fi }
od
REPLAY:
  do
  // replay messages back onto the channel,
       also with priority
  :: atomic {
    int am;
    select(am : 0 .. len(gadget_mem)-1);
    do
    :: am != 0 ->
      am = am-1;
      gadget_mem ? b_0 -> attacker_mem_0 !
          b_0;
    :: am == 0 ->
      gadget_mem ? b_0 -> c ! b_0;
      break;
    od }
  :: atomic { empty(gadget_mem) -> goto
       BREAK; }
  od
BREAK: }
```

Figure 7: Example reordering attacker model gadget with the selected replay limit as 3, targetting channel "cn"

3.3 Usage

To demonstrate the usage of PANDA, we provide a step-by-step example of proving the alternate bit protocol (ABP) is secure with respect to attackers that can replay messages. ABP is a simple communication protocol that provides reliable communication between two peers over an unreliable communication by continually agreeing on a bit value.

To use PANDA, the user first authors a PROMELA model and a correctness property in LTL. For example, take the PROMELA model as shown in Listing 2. The sender repeatedly sends its stored bit, `A_curr`, to the receiver. The receiver changes its internal bit, `B_curr`, and sends an acknowledgement to the sender. When the sender receives the acknowledgement, it will bitflip `A_curr` and repeatedly send the updated bit. A natural specification for this protocol, formalized into the LTL property `eventually_agrees`, states that if the sender and receiver do not currently agree on a bit, they eventually will be able to reach an agreement.

```
chan StoR = [2] of { bit };
chan RtoS = [2] of { bit };

bit A_curr = 0, B_curr = 1, rcv_a, rcv_b;

active proctype Sender(){
  do
  :: StoR ! A_curr;
  :: RtoS ? rcv_a ->
    if :: rcv_a == A_curr ->
      A_curr = (A_curr + 1) % 2;
    fi
  od
}

active proctype Receiver(){
  do
  :: RtoS ! B_curr;
  :: StoR ? rcv_b ->
   :: rcv_b != B_curr ->
      B_curr = rcv_b;
    fi
  od
}

ltl eventually_agrees {
  (A_curr != B_curr) implies eventually
      (A_curr == B_curr)
}
```

Listing 2: Example (simplified) PROMELA model of the alternating bit protocol.

Next, the user selects a *channel* to generate an attacker on, and an attacker model of choice. For example, we select `StoR` and `RtoS` as our channels of choice, `replay` as our attacker model of choice, and assume the ABP model is in the file `abp.pml`. Then, we run PANDA via command line.

```
$ ./panda --model=abp.pml --attacker=replay
    --channel=StoR,RtoS --eval
```

PANDA will then modify the `abp.pml` file to include the `replay` attacker gadgets attacking channels `StoR` and `RtoS`, and model-check it with SPIN. PANDA outputs the following text, cut down for readability, indicating an exhaustive search for attacks:

```
Full statespace search for:
    never claim + (eventually_agrees)

ltl eventually_agree ((A_curr!=B_curr)))
    implies (eventually ((A_curr==B_curr))

PANDA's exhaustive search is complete, no
    attacks found!
```

If desired, `-output` can also be specified so the PANDA-modified `abp.pml` can be more closely examined and modified. A full shell-script replicating this example is available in the artifact.

4 Case Studies

In this section we describe two case studies: the Transmission Control Protocol (TCP), a data transfer protocol, and Raft, a state machine replication protocol.

4.1 TCP

Transmission Control Protocol (TCP) is a transport-layer protocol designed to establish reliable, ordered communications between two peers. TCP is ubiquitous in today's internet, and therefore has seen ample formal verification efforts [12, 29, 33], including using PROMELA and SPIN [29]. We construct a TCP PROMELA model referencing the set of TCP RFCs. For our analysis, we borrow the four LTL properties used in [29], as detailed below:

ϕ_1 = No half-open connections.

ϕ_2 = Passive/active establishment eventually succeeds.

ϕ_3 = Peers don't get stuck.

ϕ_4 = `SYN_RECEIVED` is eventually followed by `ESTABLISHED`, `FIN_WAIT_1`, or `CLOSED`.

We evaluated the TCP PROMELA model against PANDA's drop, replay, and reordering attacker models on a single unidirectional communication channel. The resulting breakdown of attacks discovered is shown in Figure 8.

4.2 Raft

Raft is a consensus algorithm designed to replicate a state machine across distributed peers, and sees broad usage in distributed databases, key-value stores, distributed file systems, distributed load-balancers, and container orchestration. Historically, verification efforts of Raft using both constructive,

	Drop Attacker	Replay Attacker	Reorder Attacker
ϕ_1			
ϕ_2	x	x	
ϕ_3			
ϕ_4			

Figure 8: Automatically discovered attacks against our TCP model for ϕ_1 through ϕ_4. "x" indicates an attack was discovered, and no "x" indicates PANDA proved the absence of an attack via an exhaustive search. These experiments were ran on a laptop with an eighth generation i7 and 16gb of memory. Full attack traces are available in the artifact.

mechanized proving techniques [27,39,40] and automated verification [27] have reasoned about the protocol under certain assumptions about the stability of the communication channels. Previously, Raft has been proven to maintain properties of interest with respect volatile, attacker-controlled channels constructively using Rocq[3] [39]. However, no previous approach to Raft verification has reasoned explicitly about a coordinated, arbitrary on-channel attacker *external* to the protocol itself. Uniquely, PANDA enables us to study Raft in this context.

Referencing the original Raft thesis [27] and other raft models [40], we constructed a PROMELA model of the Raft protocol. Additionally, we derived and formalized the following properties, which our PROMELA model satisfies:

ϕ_1 = No two servers can be leaders in the same term.

ϕ_2 = Entries committed to the log at the same index must be equivalent.

ϕ_3 = Only leaders may append entires to the log.

ϕ_4 = If a leader commits at an index, any server that becomes leader afterwards must follow that commit.

ϕ_5 = If any two servers commit the same log entry, the log entry at the previous index must be equivalent

We construct our Raft model such that we can model-check an arbitrary number of peers. We also designed our model such that each peer maintains separate channels for receiving AppendEntry requests, AppendEntry responses, RequestVote requests, and RequestVote responses. This gives PANDA ample handle to reason about Raft. In particular, we study Raft in the presence of drop and replay attackers on all four aforementioned channel types, attacking both a minority and majority of peers.

To test PANDA, we altered our original Raft model to introduce a subtle bug in the raft consensus mechanism by not ensuring votes come from unique peers. We'll refer to our original, correct Raft model as `raft.pml`, and our buggy Raft model as `raft-bug.pml`. Both `raft.pml` and `raft-bug.pml` passed on ϕ_1-ϕ_5 (that is, assuming the channels are perfect). We assess `raft-bug.pml` with PANDA, and a breakdown of our findings is shown in Figure 9.

[3]Previously known as Coq

Scenario	Attack found?
Dropping AppendEntries messages	no
Dropping RequestVote messages	no
Replaying RequestVote messages	yes (ϕ_1, ϕ_4 violated)
Replaying AppendEntry messages	no
Dropping RequestVoteResponse messages	no
Dropping AppendEntryResponse messages	no

Figure 9: Breakdown of the attacker scenarios assessed with PANDA against our buggy Raft PROMELA model, `raft-bug.pml`. In all experiments, the Raft model was set to five peers and the drop/replay limits of the gadgets PANDA synthesized were set to two. We conducted our experiments on a research computing cluster, allocating 250GB of memory to each verification run. The full models and attacker traces are included in the artifact.

In our experiments, we found just one attack on our `raft-bug.pml` PROMELA model, violating election safety in particular. In this scenario, peer A and peer B are candidates for election. Peer A receives three votes, one from itself and two from other peers, and Peer B receives two votes, one from itself and one from another peer. The replay attacker simply replays the vote sent to peer B. Then, both Peer A and Peer B are convinced they won the election and change their state to leader. Following this, leader completeness is also naturally violated. In this scenario, PANDA demonstrates its ability to discover subtle bugs in protocol logic, exploiting the buggy Raft implementation.

4.3 SCTP

SCTP is a transport-layer protocol proposed as an alternative to TCP, featuring a four-way handshake, multi-homing, and multi-streaming. Among other use cases, SCTP is the data transfer protocol for various telecoms signaling protocols as well as WebRTC. For our analysis, we borrow the ten LTL properties and PROMELA models derived from the SCTP RFCs as described in [15]. We evaluated the SCTP PROMELA model against PANDA's drop, replay, and reordering attacker models on a single uni-directional communication channel. The drop attacker model was specified to max out at three dropped packets, while the replay and reordering attacker model was specified to max out at two packets. SCTP is designed to resist drop, replay, and reordering attackers [34], and we employ PANDA to exhaustively demonstrate this is the case.

5 Related Work

Similar Tools. Several formal methods tools reason about attackers on secure protocols, primarily in the cryptographic context: ProVerif, VerifPal, Tamarin, and Scyther are *Symbolic* and abstract away cryptographic primitives as terms [3, 8, 13, 24], while CryptoVerif and EasyCrypt are *computational* and reason about game-based cryptographic security proofs [7, 30]. For a general overview, see [2, 4]. Before PANDA, model checker-based approaches for reasoning about secure protocols have typically employed SPIN or TLA+ and only reasoned about correctness [11, 14, 23, 26, 38].

Reasoning About Channels. There is a long history of using formal methods tools ad-hoc to reason about on-channel attackers, particularly in the context of Byzantine protocols [10, 14, 39]. Formal methods tools have also been applied to reason about message tampering [5], delays [16], and congestion control [22]. In fact, the mathematical attacker synthesis described in [36] comes with a simple implementation in SPIN, KORG, which allows a user to manually specify messages to be *inserted* onto a victim channel. Our tool nicely extends KORG, while relieving the user of any specific manual specification.

6 Conclusion

In conclusion, PANDA addresses a critical gap in the formal verification of distributed protocols by enabling the synthesis of communication channel-based attacks against arbitrary linear temporal logic specifications. By leveraging SPIN, PANDA ensures soundness and completeness in attack synthesis. Its modular support for pre-defined attacker models enhances its versatility, enabling thorough protocol analysis across diverse and interesting scenarios. We demonstrate the effectiveness of PANDA by employing it to study TCP, Raft, and SCTP, marking it as an invaluable tool for ensuring the validity and security of distributed protocols.

References

[1] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, page 1–16, Virtual Event USA, August 2021. ACM.

[2] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)*, page 777–795, May 2021.

[3] David Basin, Cas Cremers, Jannik Dreier, and Ralf Sasse. Tamarin: Verification of large-scale, real-world, cryptographic protocols. *IEEE Security & Privacy*, 20(3):24–32, May 2022.

[4] David Basin, Cas Cremers, and Catherine Meadows. *Model Checking Security Protocols*, page 727–762. Springer International Publishing, Cham, 2018.

[5] Noomene Ben Henda. Generic and efficient attacker models in spin. In *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*, page 77–86, San Jose CA USA, July 2014. ACM.

[6] Benjamin Beurdouche. Formal verification for high assurance security software in fstar.

[7] Bruno Blanchet and Charlie Jacomme. Cryptoverif: a computationally-sound security protocol verifier.

[8] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.05: Automatic cryptographic protocol verifier, user manual and tutorial.

[9] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. De Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. *The mCRL2 Toolset for Analysing Concurrent Systems: Improvements in Expressivity and Usability*, volume 11428 of *Lecture Notes in Computer Science*, page 21–39. Springer International Publishing, Cham, 2019.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[11] Edmund M Clarke and Qinsi Wang. 25 years of model checking.

[12] Guillaume Cluzel, Kyriakos Georgiou, Yannick Moy, and Clément Zeller. Layered formal verification of a tcp stack. In *2021 IEEE Secure Development Conference (SecDev)*, page 86–93, Atlanta, GA, USA, October 2021. IEEE.

[13] Cas J. F. Cremers. *The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols*, volume 5123 of *Lecture Notes in Computer Science*, page 414–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[14] Giorgio Delzanno, Michele Tatarek, and Riccardo Traverso. Model checking paxos in spin. *Electronic Proceedings in Theoretical Computer Science*, 161:131–146, August 2014.

[15] Jacob Ginesin, Max von Hippel, Evan Defloor, Cristina Nita-Rotaru, and Michael Tüxen. A formal analysis of sctp: Attack synthesis and patch verification. (arXiv:2403.05663), March 2024. arXiv:2403.05663 [cs].

[16] Jacob Ginesin, Max von Hippel, Evan Defloor, Cristina Nita-Rotaru, and Michael Tüxen. A formal analysis of sctp: Attack synthesis and patch verification. (arXiv:2403.05663), March 2024. arXiv:2403.05663 [cs].

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, page 1–17, Monterey California, October 2015. ACM.

[18] Gerard J. Holzmann. Mars code. *Communications of the ACM*, 57(2):64–73, February 2014.

[19] Gerard J. Holzmann and Margaret H. Smith. Automating software feature verification. *Bell Labs Technical Journal*, 5(2):72–87, 2000.

[20] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[21] Chiao Hsieh and Sayan Mitra. *Dione: A Protocol Verification System Built with Dafny for I/O Automata*, volume 11918 of *Lecture Notes in Computer Science*, page 227–245. Springer International Publishing, Cham, 2019.

[22] Samuel Jero, Endadul Hoque, David Choffnes, Alan Mislove, and Cristina Nita-Rotaru. Automated attack discovery in tcp congestion control using a model-guided approach. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[23] Abdul Sahid Khan, Madhavan Mukund, and S. P. Suresh. *Generic Verification of Security Protocols*, volume 3639 of *Lecture Notes in Computer Science*, page 221–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[24] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. Verifpal: Cryptographic protocol analysis for the real world.

[25] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, page 254–266, Providence, RI, USA, September 1977. IEEE.

[26] Prasad Narayana, Ruiming Chen, Yao Zhao, Yan Chen, Zhi Fu, and Hai Zhou. Automatic vulnerability checking of ieee 802.16 wimax protocols through tla+. In *2006 2nd IEEE Workshop on Secure Network Protocols*, page 44–49, November 2006.

[27] Diego Ongaro. Consensus: Bridging theory and practice.

[28] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm.

[29] Maria Leonor Pacheco, Max Von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. Automated attack synthesis by extracting finite state machines from protocol specification documents. In *2022 IEEE Symposium on Security and Privacy (SP)*, page 51–68, San Francisco, CA, USA, May 2022. IEEE.

[30] Vitor Pereira. Easycrypt - a (brief) tutorial.

[31] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*, volume 10801 of *Lecture Notes in Computer Science*, page 619–650. Springer International Publishing, Cham, 2018.

[32] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, January 2018.

[33] Mark Anthony Shawn Smith. *Formal verification of TCP and T/TCP*. Thesis, Massachusetts Institute of Technology, 1997. Accepted: 2008-09-03T18:09:43Z.

[34] M. Tüxen, R. Stewart, K. Nielsen, R. Jesup, and S. Loreto. Stream Control Transmission Protocol (SCTP) Specification Errata and Issues. Request for Comments, June 2022.

[35] W. Visser, K. Havelund, G. Brat, and Seungjoon Park. Model checking programs. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, page 3–11, Grenoble, France, 2000. IEEE.

[36] Max von Hippel, Cole Vick, Stavros Tripakis, and Cristina Nita-Rotaru. Automated attacker synthesis for distributed protocols. (arXiv:2004.01220), April 2022. arXiv:2004.01220 [cs].

[37] Hillel Wayne. Tla+ message passing, October 2018.

[38] Hillel Wayne. Modeling adversaries with tla+. `https://www.hillelwayne.com/post/adversaries/`, 2019. Accessed: 2024-12-03.

[39] James R Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems.

[40] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*, page 154–165, St. Petersburg FL USA, January 2016. ACM.

7 Appendix

7.1 Mathematical Preliminaries

Linear Temporal Logic (LTL) is a model logic for reasoning about program executions. In LTL, we say a program P *models* a property ϕ (notationally, $P \models \phi$). That is, ϕ holds over every execution of P. If ϕ does not hold over every execution of P, we say $P \not\models \phi$. The LTL language is given by predicates over a first-order logic with additional temporal operators: *next*, *always*, *eventually*, and *until*.

An LTL model checker is a tool that, given P and ϕ, can automatically check whether or not $P \models \phi$; in general, LTL is a *decidable* logic, and LTL model checkers will always be able to decide whether $P \models \phi$ given enough time and resources.

We use $||$ to denote rendezvous composition. That is, if $S = P \,||\, Q$, processes P and Q are composed together into a singular state machine by matching their equivalent transitions.

LTL program synthesis is the problem of, given an LTL specification ϕ, automatically deriving a program P that satisfies ϕ (that is, $P \models \phi$). *LTL attack synthesis* is logically dual to LTL program synthesis. In attack synthesis, the problem is flipped: given a program P and a property ϕ such that $P \models \phi$, we ask whether there exists some "attack" A such that $(P \,||\, A) \not\models \phi$. Fundamentally, PANDA is a synthesizer for such an A, based upon the distributed systems attacker framework as described in [36].

7.2 Arguments for PANDA's Soundness, Completeness, and Complexity

As aforementioned, PANDA is an extended implementation of the theoretical attack synthesis framework proposed by [36]. This framework enjoys soundness and completeness guarantees for attacks discovered; that is, if there exists an attack, it is discovered, and if an attack is discovered, it is valid. However, the attack synthesis framework proposed by [36] reasons about an abstracted, theoretical process construct. Therefore, in order to correctly claim PANDA is also sound and complete, it is necessary to demonstrate discovering an attack within the theoretical framework reduces to the semantics of SPIN, the model checker PANDA is built on top of.

Definition 1 (Büchi Automata). *A Büchi Automata is a tuple $B = (Q, \Sigma, \delta, Q_0, F)$ where:*

- *Q is a finite set of states,*
- *Σ is a finite alphabet,*
- *$\delta \subseteq Q \times \Sigma \times Q$ is a transition relation,*
- *$Q_0 \subseteq Q$ is a set of initial states,*
- *$F \subseteq Q$ is a set of accepting states.*

A run of a Büchi Automata is an infinite sequence of states $q_0, q_1, q_2, \ldots$ such that $q_0 \in Q_0$ and $(q_i, a, q_{i+1}) \in \delta$ for some $a \in \Sigma$ at each step i. The run is considered accepting if it visits states in F infinitely often.

Definition 2 (Process). *A* Process *is a tuple $P = \langle AP, I, O, S, s_0, T, L \rangle$, where:*

- *AP is a finite set of atomic propositions,*
- *I is a set of inputs,*
- *O is a set of output, such that $I \cap O = \emptyset$,*
- *S is a finite set of states,*
- *$s_0 \in S$ is the initial state,*
- *$T \subseteq S \times (I \cup O) \times S$ is the transition relation,*
- *$L : S \to 2^{AP}$ is a labeling function mapping each state to a subset of atomic propositions.*

A transition $(s, x, s') \in T$ is called an input transition *if $x \in I$ and an* output transition *if $x \in O$.*

Theorem 1. *A process, as defined in [36], always directly corresponds to a Büchi Automata.*

Proof. Given a Büchi Automata $B = (Q, \Sigma, \delta, Q_0, F)$, we construct a corresponding Process $P = \langle AP, I, O, S, s_0, T, L \rangle$ as follows:

- Atomic Propositions: $AP = \{\text{accept}\}$, a singleton set containing a special proposition indicating acceptance.
- Inputs and Outputs: $I = \Sigma$ and $O = \emptyset$.
- States: $S = Q$ and $s_0 \in Q_0$.
- Transition Relation: $T = \delta$.
- Labeling Function: $L : S \to 2^{AP}$ defined by

$$L(s) = \begin{cases} \{\text{accept}\} & \text{if } s \in F, \\ \emptyset & \text{otherwise.} \end{cases}$$

In this mapping, the states and transitions of the BA are preserved in the Process, and the accepting states F are identified via the labeling function L.

Conversely, given a Process $P = \langle AP, I, O, S, s_0, T, L \rangle$ with an acceptance condition defined by a distinguished proposition $p \in AP$, we define a Büchi Automata $B = (Q, \Sigma, \delta, Q_0, F)$ as follows:

- States: $Q = S$ and $Q_0 = \{s_0\}$.
- Alphabet: $\Sigma = I \cup O$.
- Transition Relation: $\delta = T$.

- Accepting States: $F = \{s \in S \mid p \in L(s)\}$.

Here, the accepting states in the BA correspond to those states in the Process that are labeled with the distinguished proposition p.

In both structures, a run is an infinite sequence of states connected by transitions:

- In the Büchi Automata: $q_0, q_1, q_2, \ldots$ with $q_0 \in Q_0$ and $(q_i, a_i, q_{i+1}) \in \delta$ for some $a_i \in \Sigma$.
- In the Process: $s_0, s_1, s_2, \ldots$ with $s_0 = s_0$ and $(s_i, x_i, s_{i+1}) \in T$ for some $x_i \in I \cup O$.

An accepting run in the Büchi Automata visits states in F infinitely often. Similarly, an accepting run in the Process visits states labeled with p infinitely often. Since $F = \{s \in S \mid p \in L(s)\}$, the acceptance conditions are preserved under the mappings. □

Definition 3 (Threat Model)**.** *A threat model is a tuple* $(P, (Q_i)_{i=0}^{m}, \phi)$ *where:*

- $P, Q_0, \ldots, Q_m$ *are processes.*
- *Each process* Q_i *has no atomic propositions (i.e., its set of atomic propositions is empty).*
- φ *is an LTL formula such that* $P \parallel Q_0 \parallel \cdots \parallel Q_m \models \phi$.
- *The system* $P \parallel Q_0 \parallel \cdots \parallel Q_m$ *satisfies the formula* ϕ *in a non-trivial manner, meaning that* $P \parallel Q_0 \parallel \cdots \parallel Q_m$ *has at least one infinite run.*

Theorem 2. *Checking whether there exists an attacker under a given threat model, the R-∃ASP problem as proposed in [36], is equivalent to Büchi Automata language inclusion (which is in turn solved by the* SPIN *model checker).*

Proof. For a given threat model $(P, (Q_i)_{i=0}^{m}, \phi)$, checking $\exists ASP$ is equivalent to checking

$$R = MC(P \,||\, \text{Daisy}(Q_0) \,||\, \ldots \,||\, \text{Daisy}(Q_m), \phi)$$

Where MC is a model checker, and Daisy(Q_i) is for intents of this proof, equivalent to a process. Therefore, via the previous theorem we can construct Büchi Automata $BA_P, BA_{\text{Daisy}(Q_0)}, \ldots, BA_{\text{Daisy}(Q_m)}$ from the processes $P, \text{Daisy}(Q_0), \ldots, \text{Daisy}(Q_m)$. Then, we check

$$\text{SPIN}(BA_P \,||\, BA_{\text{Daisy}(Q_0)} \,||\, \ldots \,||\, BA_{\text{Daisy}(Q_m)}, \phi)$$

Or equivalently, translating ϕ to the equivalent Büchi Automata BA_ϕ via [20], we equivalently check

$$\left(BA_P \,||\, BA_{\text{Daisy}(Q_0)} \,||\, \ldots \,||\, BA_{\text{Daisy}(Q_m)}\right) \subseteq BA_\phi$$

Where rendezvous composition for I/O Büchi Automata is precise the same as for I/O Kripke Automata; that is, input and output transitions are matched. It's easy to see these composition operations are equivalent. □

Theorem 3. *Checking whether there exists an attacker for a given threat model, the R-∃ASP problem as proposed in [36], is in PSPACE.*

Proof. By the previous argument the ∃ASP problem corresponds to Büchi Automata language inclusion, which is well-known to be PSPACE-complete [25]. □

7.3 Priorities & On-the-fly Büchi Automata Composition

As described in Appendix Section 7.2, SPIN reduces verification problems to deciding Büchi Automata intersection emptiness. That is, given n Büchi Automata programs $P_1, \ldots, P_n$, SPIN decides:

$$L(P_1) \cap L(P_2) \cap \ldots \cap L(P_n) = \emptyset$$

Or equivalently:

$$\exists w \in \Sigma \text{ such that } w \in \bigcap_{i=1}^{n} L(P_i)$$

The canonical decision procedure for deciding intersection emptiness involves constructing a composite automata $P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n$ whose acceptance states are the intersection of each process's acceptance states. Then, this composite automata is exhaustively searched. This method is infeasible in practice, as it requires building the entire automata – which is $|P_1| \times \ldots \times |P_n|$ in size – in memory.

Therefore, every practical intersection emptiness implementation employs *on-the-fly composition* to search the composite automata without explicitly constructing it in memory. Recall the is defined as a 5-tuple (as defined in Appendix Section 7.2), $(Q, \Sigma, \delta, Q_0, F)$. The state of the composite automata is denoted as $(q^{(P_1)}, \ldots, q^{(P_n)})$ where $q^{(P_i)} \in Q^{(P_i)}$, for $1 \leq i \leq n$. Similarly, the transition function becomes:

$$\delta : \left((\delta^{(P_1)} : Q^{(P_1)} \times \Sigma) \times \ldots \times (\delta^{(P_n)} : Q^{(P_n)} \times \Sigma)\right) \\ \rightarrow (Q^{(P_1)} \times \ldots \times Q^{(P_n)})$$

Such that only one such $\delta^{(P_i)}$ can be invoked per transition of δ. Therefore, *priorities* in SPIN are implemented by always invoking the highest priority $\delta^{(P_i)}$ among all the possible process transition functions that can be invoked.