PANDA: An Attack Synthesis Tool for Distributed Protocols

***Abstract*—Distributed protocols underpin the modern internet, making their correctness and security critical. Formal methods provide rigorous tools for analyzing protocol correctness and cryptographic security, yet existing tools fall short for denial of service (DoS) analysis. We introduce PANDA, a tool that synthesizes attacks on distributed protocols by targeting communication channels to violate linear temporal logic (LTL) specifications. PANDA provides sound, complete analysis, synthesizing attacks or proving their absence through exhaustive state-space search. With support for pre-defined and custom attacker models, PANDA enables targeted DoS analysis and broader LTL-based verification, demonstrated through various case studies.**

***Index Terms*—Protocols, Attack Synthesis, Denial of Service, Model Checking**

I. Introduction

Distributed protocols are the foundation for the modern internet, and therefore ensuring their correctness and security is paramount. To this end, formal methods, the use of mathematically rigorous techniques for reasoning about software, has been increasingly employed to analyze and study distributed protocols. Historically, formal methods has been employed for reasoning about concurrency and distributed algorithms [1]–[3], and in recent years formal methods have been employed at scale to reason about the security of cryptographic protocols and primitives [4]–[8]. This myriad of formal methods tooling applicable to secure protocols has enabled reasoning about security-relevant properties involving secrecy, authentication, indistinguishability in addition to concurrency, safety, and liveness. However, no previous formal methods tooling offered an effective solution for rigorously studying an attacker that controls communication channels. That is, how do you reason about an attacker that can arbitrarily drop, reorder, replay, or insert messages onto a communication channel?

To fill this gap, we introduce PANDA[1], a tool for synthesizing attacks on distributed protocols that implements and extends the theoretical framework proposed in [9]. In particular, PANDA targets the communication channels between the protocol endpoints, and synthesizes attacks to violate arbitrary linear temporal logic (LTL) specifications. PANDA either synthesizes attack, or proves 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 supports pre-defined attacker models, including attackers that can replay, reorder, or drop messages on channels, as well as custom user-defined attacker models. Although PANDA best lends itself for reasoning about denial of service attacks, it can target any specification expressable in LTL. We present several case studies illustrating the usefulness of PANDA. We release our code as our models as open source at Cristina says: here add the anonymous link.

[1]PANDA is a fictitious name for our system, for double-blind submission.

II. PANDA Architecture

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

A. High-level design

Cristina says: need introductory paragraph about program synthesis, the main idea

At the highest level, PANDA sits on user-specified communication channels in a program written in PROMELA, the modeling language of the SPIN model checker. The user selects an attacker model of choice and correctness properties of choice. PANDA then invokes SPIN, which exhaustively searches for attacks with respect to the chosen attacker model, PROMELA model, and correctness property. A high-level overview of the PANDA pipeline is given in the Figure 1.

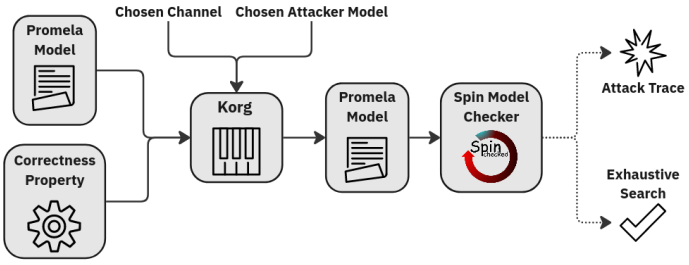

Fig. 1. A high-level overview of the PANDA workflow

B. Supported Attacker Models

PANDA supports four general attacker model gadgets: an attacker that can drop, replay, reorder, or insert messages on a channel. In this section we discuss the various details that went into the implementation of the gadgets that encapsulate the behavior of the respective attacker models.

Drop Attacker Model Gadget The most simple attacker model PANDA supports is an attacker that can *drop* messages from a 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. An example is shown in Figure 1.

Replay Attacker Model Gadget The next attacker model PANDA supports is an attacker that can observe and *replay* 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 model gadget PANDA employs 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 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.

Reorder Attacker Model Gadget PANDA supports synthesizing attackers that can *reorder* 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 reordering attacker model gadget PANDA synthesizes works as follows. 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.

Insert Attacker Models PANDA supports the synthesis of attackers that can simply insert messages onto a channel. While the drop, replay, and reordering attacker model gadgets as previously described have complex gadgets that PANDA synthesizes with respect to a user-specified channel, the insert attacker model gadget is synthesized with respect to a user-defined *IO-file*. This file denotes the specific outputs and channels the attacker is capable of sending, and PANDA generates a gadget capable of synthesizing attacks using the given inputs. An example I/O file is given in Figure 4, and the generated gadget is given in Figure 5.

These attacker models can be mixed and matched as desired by the PANDA user. For example, a user can specify a drop attacker and replay attacker to target channel 1, a reordering attacker to target channel 2, and an insert attacker to target channel 3. If multiple attacker models are declared, PANDA will synthesize attacks where the attackers on different channel *coordinate* to construct a unifying attack.

```
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:
}
```

Listing 1. Example dropping attacker model gadget with drop limit of 3, targetting channel "cn"

C. PANDA *Implementation*

We implemented PANDA on top of the SPIN, a popular and robust model checker for reasoning about distributed and concurrent systems. 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 6. 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.

D. Usage

To demonstrate the usage of PANDA, we provide a step-by-step example of proving the alternate bit protocol (ABP)

```
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 i = 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;
    i--;
    if
    :: i == 0 -> goto REPLAY;
    :: i != 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 use all messages on the
      channel
  :: 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:
}
```

Listing 2. Example replay attacker model gadget with the selected replay limit as 3, targetting channel "cn"

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

chan gadget_mem = [3] of { int, int, int };
active proctype attacker_reordering()
    priority 255 {
byte b_0, b_1, b_2, blocker;
int i = 3;
INIT:
do
  // arbitrarily choose a message to start
      consuming on
  :: {
      blocker = len(cn);
      do
      :: b != len(c) -> goto INIT;
      od
    }
  :: goto CONSUME;
od
CONSUME:
do
  // consume messages with high priority
  :: c ? [b_0] -> atomic {
    c ? b_0 -> gadget_mem ! b_0;
    i--;
    if
    :: i == 0 -> goto REPLAY;
    :: i != 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:
}
```

Listing 3. Example reordering attacker model gadget with the selected replay limit as 3, targetting channel "cn"

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 7. 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 };
```

```
cn:
    I:
    O:1-1-1, 1-2-3, 3-4-5
```

Listing 4. Example I/O file targetting channel "cn"

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

active proctype daisy() {
INIT:
  do
  :: cn ! 1,1,1;
  :: cn ! 1,2,3;
  :: cn ! 3,4,5;
  :: goto RECOVERY;
  od
RECOVERY:
}
```

Listing 5. Example gadget synthesized from an I/O file targetting the channel "cn"

```
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 7. 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.

III. CASE STUDIES

In this section we describe two case study, TCP transport protocol and RAFT state machine replication protocol.

A. TCP

TCP (Transmission Control Protocol) 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 [11]–[13], including using PROMELA and SPIN [13]. We construct a TCP PROMELA model referencing the set of TCP RFCs. For our analysis, we borrow the four LTL properties used in [13], 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 III-A.

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

Fig. 2. Automatically discovered attacks against the hand-written TCP model from Pacheco et al. and our own, 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. Full attack traces are available in the artifact.

B. 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,

mechanized proving techniques [15]–[17] and automated verification [17] have reasoned about the protocol under certain assumptions about the stability of the communication channels. However, no previous approach to Raft verification has reasoned about an 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 [17] and other raft models [15], we constructed a PROMELA model of the Raft protocol. Additionally, we derived and formalized the following properties, which our PROMELA model satisfies:

$\phi_1 =$ No two servers can be leaders in the same term.

$\phi_2 =$ Entries committed to the log at the same index must be equivalent.

$\phi_3 =$ Only leaders may append entires to the log.

$\phi_4 =$ If a leader commits at an index, any server that becomes leader afterwards must follow that commit.

$\phi_5 =$ 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. A breakdown of our findings is shown in Figure **??**.

IV. 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 and Raft, marking it as an invaluable tool for ensuring the validity and security of distributed protocols.

REFERENCES

[1] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, p. 872–923, May 1994.

[2] G. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, p. 279–295, May 1997.

[3] E. M. Clarke and Q. Wang, "25 years of model checking."

[4] D. Basin, C. Cremers, J. Dreier, and R. Sasse, "Tamarin: Verification of large-scale, real-world, cryptographic protocols," *IEEE Security & Privacy*, vol. 20, no. 3, p. 24–32, May 2022.

[5] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "Proverif 2.05: Automatic cryptographic protocol verifier, user manual and tutorial."

[6] N. Kobeissi, G. Nicolas, and M. Tiwari, "Verifpal: Cryptographic protocol analysis for the real world."

[7] B. Blanchet and C. Jacomme, "Cryptoverif: a computationally-sound security protocol verifier."

[8] D. Basin, F. Linker, and R. Sasse, "A formal analysis of the imessage pq3 messaging protocol."

[9] Anonym, "Anonymized for blinded submission," XXX.

[10] D. Kozen, "Lower bounds for natural proof systems," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. Providence, RI, USA: IEEE, Sep. 1977, p. 254–266. [Online]. Available: http://ieeexplore.ieee.org/document/4567949/

[11] G. Cluzel, K. Georgiou, Y. Moy, and C. Zeller, "Layered formal verification of a tcp stack," in *2021 IEEE Secure Development Conference (SecDev)*. Atlanta, GA, USA: IEEE, Oct. 2021, p. 86–93. [Online]. Available: https://ieeexplore.ieee.org/document/9652642/

[12] M. A. S. Smith, "Formal verification of tcp and t/tcp," Thesis, Massachusetts Institute of Technology, 1997, accepted: 2008-09-03T18:09:43Z. [Online]. Available: https://dspace.mit.edu/handle/1721.1/42779

[13] M. L. Pacheco, M. V. Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, "Automated attack synthesis by extracting finite state machines from protocol specification documents," in *2022 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2022, p. 51–68. [Online]. Available: https://ieeexplore.ieee.org/document/9833673/

[14] M. von Hippel, C. Vick, S. Tripakis, and C. Nita-Rotaru, "Automated attacker synthesis for distributed protocols," no. arXiv:2004.01220, Apr. 2022, arXiv:2004.01220 [cs]. [Online]. Available: http://arxiv.org/abs/2004.01220

[15] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. 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*. St. Petersburg FL USA: ACM, Jan. 2016, p. 154–165. [Online]. Available: https://dl.acm.org/doi/10.1145/2854065.2854081

[16] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson, "Verdi: A framework for implementing and formally verifying distributed systems."

[17] D. Ongaro, "Consensus: Bridging theory and practice."