KORG: An Attack Synthesizer for Distributed Protocols

Jacob Ginesin
Northeastern University

Max von Hippel
Northeastern University

Cristina Nita-Rotaru
Northeastern University

***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 KORG, a tool that synthesizes attacks on distributed protocols by targeting communication channels to violate linear temporal logic (LTL) specifications. KORG provides sound, complete analysis, synthesizing attacks or proving their absence through exhaustive state-space search. With support for pre-defined and custom attacker models, KORG 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 KORG, a tool for synthesizing attacks on distributed protocols that implements the theoretical framework proposed in Hippel et al. [9]. In particular, KORG targets the communication channels between the protocol endpoints, and synthesizes attacks to violate arbitrary linear temporal logic (LTL) specifications. KORG either synthesizes attack, or proves the absence of such via an exhaustive state-space search. KORG is sound and complete, meaning if there exists an attack KORG will find it, and KORG will never have false positives. KORG 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 KORG best lends itself for reasoning about denial of service attacks, it can target any specification expressable in LTL. We present a variety of case studies illustrating the employability and usefulness of KORG.

II. DESIGN METHODOLOGY

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

A. High-level design

At the highest level, KORG 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. KORG 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 KORG pipeline is given in the Figure 1.

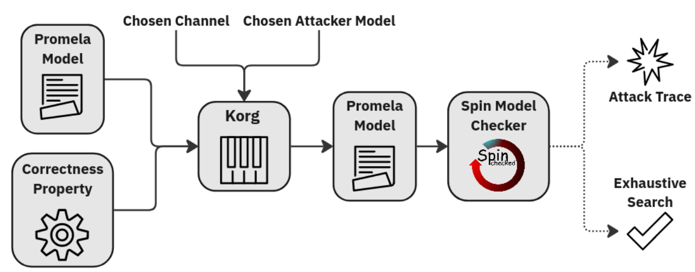

Fig. 1. A high-level overview of the KORG workflow

B. Supported Attacker Models

KORG supports the automatic synthesis of attacks with respect to four general pre-defined attacker models applicable to any communication channel:

- **Drop Attacker Model**. Drop attackers are capable of dropping a finite number of messages off a channel.
- **Replay Attacker Model**. Replay attackers are capable of replaying previously seen messages back onto a channel.
- **Reorder Attacker Model**. Reorder attackers are capable of reordering messages on a channel.
- **Insert Attacker Model**. Insert attackers are capable of inserting arbitrary messages (as specifiable by the user) onto a channel.

These attacker models can be mixed and matched as desired by the KORG 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, KORG will synthesize attacks where the attackers on different channel *coordinate* to construct a unifying attack.

C. Soundness And Completeness of Korg

Fundamentally, the theoretical framework that KORG implements proposed by Hippel et al. reasons about *communicating processes*; similarly, KORG is best understood as a synthesizer for attackers that sit *between* communicating processes.

The theoretical attack synthesis framework and KORG use slightly different formalisms. Both employ derivations the general *Input/Output (I/O) automata*, state machines whose transitions indicate sending or receiving a message.[1] In particular, the theoretical attack synthesis framework defines their own notion of a *process* and argues their attack synthesis algorithm maintains soundness and completeness guarantees with respect to it, while KORG relies upon SPIN's preferred model checking formalism, the Büchi Automata. Both utilize linear temporal logic as their specification language of choice.

We ultimately seek to conclude KORG maintains the guarantees of the theoretical framework it implements, therefore it is necessary to demonstrate the equivalence of *processes* from the theoretical attack synthesis framework with the Büchi Automata. For ease of reading and clarity, we only provide shortened narrations of the arguments here. The detailed, definitions, theorems, and proofs are provided in Appendix Section VI-A.

Theorem 1. *A process, as defined in Hippel et al., always directly corresponds to a Büchi Automata.*

In short, a process in the theoretical attack synthesis framework is a Kripke Structure equipped with input and output transitions. That is, when composing two processes, an output transition must be matched to a respective input transition. Processes also include atomic propositions, which the given linear temporal logic specifications are defined over. We invoke and build on the well-known correspondence between Kripke Structures and Büchi Automata to show our desired correspondence.

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

Via the previous theorem, we can translate the threat model processes and the victim processes to Büchi Automata and intersect them. Büchi Automata intersection corresponds with Büchi Automata language inclusion, which is in turn solved by SPIN. From this result, we naturally get a complexity-theoretic result for finding an attacker from a given threat model.

Theorem 3. *Checking whether there exists an attacker for a given threat model, the R-∃ASP problem as proposed in Hippel et al., is PSPACE-complete.*

By the previous argument the attack synthesis problem reduces to intersecting multiple Büchi Automata (or alternatively Büchi Automata language inclusion), which is well-known to be PSPACE-complete [10]. Although this result implies KORG has a rough upper bound complexity, in practice due the various implementation-level optimizations of SPIN finding attacks on some property is generally fast, but proving their absence via a state-space search can expensive [3].

Since KORG uses SPIN as its underlying model checker, we can effectively conclude KORG is sound and complete.

D. The Korg Implementation

We implemented KORG 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. KORG 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.

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

E. Usage

To demonstrate the usage of KORG, we'll walk through an 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 KORG, the user first authors a PROMELA model and a correctness property in LTL. For example, take the PROMELA

[1] A fundamental assumption both KORG and the theoretical attack synthesis framework rely upon is unicast transition relations of I/O automata within this context. That is, if one sending automata has an output transition matching an input transition of two receiving automata, only one input/output transition pair can be composed upon. Model checkers for I/O automata such as SPIN will explore both possibilities.

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 KORG via command line.

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

KORG will then modify the `abp.pml` file to include the `replay` attacker gadgets attacking channels `StoR` and `RtoS`, and model-check it with SPIN. KORG 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))

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

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

III. ATTACKER MODEL GADGETS

KORG 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.

A. Drop Attacker Model Gadget

The most simple attacker model KORG 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 KORG 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 4.

B. Replay Attacker Model Gadget

The next attacker model KORG 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 KORG 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 5.

C. Reorder Attacker Model Gadget

KORG 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 KORG 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 6.

D. Insert Attacker Models

KORG 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 KORG 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 KORG generates a gadget capable of synthesizing attacks using the given inputs. An example I/O file is given in Figure 7, and the generated gadget is given in 8.

IV. CASE STUDIES

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]. A previous version of KORG has been applied TCP in [9], [13]; in particular, we study our KORG extensions using the hand-written TCP PROMELA model from [13]. Additionally, 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:

$\phi_1 =$ No half-open connections.

$\phi_2 =$ Passive/active establishment eventually succeeds.

$\phi_3 =$ Peers don't get stuck.

$\phi_4 =$ `SYN_RECEIVED` is eventually followed by `ESTABLISHED`, `FIN_WAIT_1`, or `CLOSED`.

We evaluated the our TCP PROMELA model and the hand-written TCP PROMELA model presented by [13] against KORG's drop, replay, and reordering attacker models on a single uni-directional communication channel. The resulting breakdown of attacks discovered is shown in Figure IV-A.

	Drop Attacker		Replay Attacker		Reorder Attacker	
	Pacheco et al.	Ours	Pacheco et al.	Ours	Pacheco et al.	Ours
ϕ_1						
ϕ_2	x	x	x	x		
ϕ_3						
ϕ_4					x	

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 KORG 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 [14]–[16] and automated verification [16] 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, KORG enables us to study Raft in this context.

Referencing the original Raft thesis [16] and other raft models [14], 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 KORG 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 **??**.

V. CONCLUSION

In conclusion, KORG 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, KORG 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 KORG 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] 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
[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] 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
[15] 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.”
[16] D. Ongaro, “Consensus: Bridging theory and practice.”

VI. Appendix

A. Full Korg Soundness and Completeness Proofs

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 Hippel et al., 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 Hippel et al., 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

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

Where MC is a model checker, and $\text{Daisy}(Q_i)$ 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

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

Or equivalently, translating ϕ to the equivalent Büchi Automata BA_ϕ via [2], we equivalently check

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

□

Theorem 3. *Checking whether there exists an attacker for a given threat model, the R-∃ASP problem as proposed in Hippel et al., is PSPACE-complete.*

Proof. By the previous argument the ∃ASP problem corresponds to Büchi Automata language inclusion, which is well-known to be PSPACE-complete [10]. □

B. Preventing Korg 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 KORG attacker model gadget against a PROMELA model and a liveness property, it is crucial to ensure the gadget has no cyclic behavior. If a KORG gadget has cyclic behavior in any way, it'll trivially violate the liveness property and produce a garbage attack trace. To prevent this, we make the following considerations.

First, we design our KORG gadgets such that they never arbitrarily send and consume messages to a single channel. Second, we allow KORG gadgets, which are always processing messages on channels, to arbitrarily "skip" messages on a channel if need be. To demonstrate the latter, consider the extension of the drop attacker model gadget in Figure 3. 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.

C. Attacker Model Gadget Examples

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

active proctype attacker_drop() {
int b_0, b_1, b_2, blocker;
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
    }
  // pass over a message on a channel as
     needed
  :: cn ? [b_0, b_1, b_2] -> atomic {
    // wait for the channel to change lengths
    // then, once it does, go to MAIN
    blocker = len(cn);
    do
    :: blocker != len(cn) -> goto MAIN;
    od
    }
  :: goto BREAK;
  od
BREAK:
}
```

Listing 3. Example dropping attacker model gadget with message skipping

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

```
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 5. 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 6. Example reordering attacker model gadget with the selected replay limit as 3, targetting channel ”cn”

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

Listing 7. 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 8. Example gadget synthesized from an I/O file targetting the channel ”cn”