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, rearrange, 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, rearrange, 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 a user-defined channel 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 the SPIN, which exhaustively searches for attacks with respect to the chosen model and properties. A high-level overview of the KORG pipeline is given in the Figure 1.

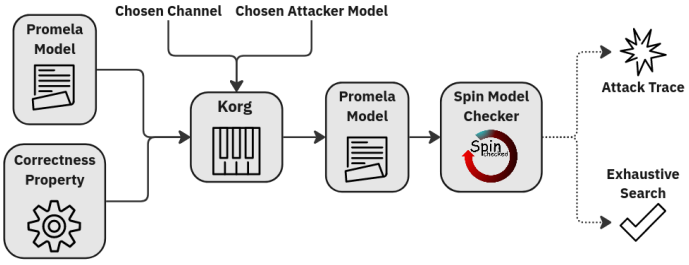

Fig. 1. A high-level overview of the KORG workflow

B. 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 attack synthesis framework proposed by Hippel et al. and KORG use slightly different formalisms. Both employ derivations the general *input/output automata*, state machines whose transitions indicate sending or receiving a message. In particular, the framework proposed by Hippel et al. defines their own notion of a *process* and argues their attack synthesis framework 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 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 Hippel et al. with the Büchi Automata. For ease of reading and clarity, we only provide the shortened arguments here. The detailed 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 as defined in Hippel et al. 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 Automatato 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 Automataand intersect them. Büchi Automata intersection corresponds with Büchi Automatalanguage 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 Automatalanguage inclusion), which is well-known to be PSPACE-complete [12]. 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.

C. 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.

Following the gadgetry framework as described in Hippel et al., 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. Additionally, users can explicitly define which messages a generated gadget can send and receive. 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.

```
// 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

D. Usage

To use KORG, the user inputs a PROMELA model, a correctness property specified in LTL, a channel from the given PROMELA model, and an attacker model of choice. KORG will then generate an attacker model gadget corresponding to the selected attacker model with respect to the chosen channel. The attacker model gadget is then appended onto the given PROMELA model and evaluated against the LTL property with SPIN. KORG will then either produce an attack trace demonstrating the precise actions the attacker took to violate the LTL property, or demonstrate the absence of an attack via an exhaustive state-space search.

Precise details of how to use the KORG implementation are provided in the anonymous repository: (link)

III. ATTACKER MODELS

KORG supports three general attacker models: an attacker that can drop, replay, or rearrange messages on a channel. Additionally, KORG supports user-defined attacker that insert arbitrary messages onto a channel. In this section we discuss the various details that go into each attacker model.

A. Dropping Attacker Model

The first and most simple general 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.

B. Replaying Attacker Model

The second 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

```
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 2. Example dropping attacker model gadget

of messages the attacker can replay back onto the specified channel.

Jake says: todo: describe impl more

C. Rearranging Attacker Model

Lastly, KORG supports an attacker model such that an attacker can *rearrange* messages on a channel. Like the drop and replay attacker models, the user can specify a "rearrange limit" that caps the number of messages that can be rearranged by the attacker on the specified channel.

D. Custom Attacker Models

While the drop, replay, and rearrange attacker models as previously described have complex gadgets that KORG synthesizes with respect to a user-specified channel, KORG also supports the synthesis of gadgets with respect to user-defined inputs and outputs.

IV. CASE STUDIES

A. SCTP

B. TCP

C. DCCP

V. CONCLUSION

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] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification." IEEE Computer Society, 1986. [Online]. Available: https://orbi.uliege.be/handle/2268/116609

[11] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 2000.

[12] 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/

VI. APPENDIX

A. Full Korg Soundness and Completeness Proofs

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

Proof. Jake says: highly standard equivalence argument □

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. Jake says: arguing the equivalence of buchi automata intersection and process composition □

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. Jake says: cite lower bounds for natural proof systems □

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.

```
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