Automated Attacker Synthesis for Distributed Protocols

Max von Hippel, Cole Vick, Stavros Tripakis, and Cristina Nita-Rotaru

Northeastern University, Boston MA 02118, USA
{vonhippel.m,stavros,c.nitarotaru}@northeastern.edu
vick.c@husky.neu.edu

Abstract. Distributed protocols should be robust to both benign malfunction (e.g. packet loss or delay) and attacks (e.g. message replay). In this paper we take a formal approach to the automated synthesis of attackers, i.e. adversarial processes that can cause the protocol to malfunction. Specifically, given a formal *threat model* capturing the distributed protocol model and network topology, as well as the placement, goals, and interface of potential attackers, we automatically synthesize an attacker. We formalize four attacker synthesis problems - across attackers that always succeed versus those that sometimes fail, and attackers that may attack forever versus those that may not - and we propose algorithmic solutions to two of them. We report on a prototype implementation called KORG and its application to TCP as a case-study. Our experiments show that KORG can automatically generate well-known attacks for TCP within seconds or minutes.

Keywords: Synthesis · Security · Distributed Protocols

1 Introduction

Distributed protocols represent the fundamental communication backbone for all services over the Internet. Ensuring the correctness and security of these protocols is critical for the services built on top of them [9]. Prior literature proposed different approaches to correctness assurance, e.g. testing [26, 12], or structural reasoning [11]. Many such approaches rely on manual analysis or are ad-hoc in nature.

In this paper, we take a systematic approach to the problem of security of distributed protocols, by using formal methods and synthesis [10]. Our focus is the automated generation of *attacks*. But what exactly is an attack? The notion of an attack is often implicit in the formal verification of security properties: it is a counterexample violating some security specification. We build on this idea. We provide a formal definition of *threat models* capturing the distributed protocol model and network topology, as well as the placement, goals, and capabilities of potential *attackers*. Intuitively, an attacker is a process that, when composed with the system, results a protocol property violation.

By formally defining attackers as processes, our approach has several benefits: first, we can ensure that these processes are *executable*, meaning attackers are programs that reproduce attacks. This is in contrast to other approaches that generate a *trace* exemplifying an attack, but not a program producing the attack, e.g. [5, 39]. Second, an explicit formal attacker definition allows us to distinguish different types of attackers, depending on: what exactly does it mean to violate a property (in some cases? in all cases?);

how the attacker can behave, etc. We distinguish between $\exists$-attackers (that sometimes succeed in violating the security property) and $\forall$-attackers (that always succeed); and between attackers *with recovery* (that eventually revert to normal system behavior) and attackers without (that may attack forever). We make four primary contributions.

- We propose a novel formalization of threat models and attackers, where the threat models algebraically capture not only the attackers but also the attacker goals, the environmental and victim processes, and the network topology.
- We formalize four attacker synthesis problems – $\exists$ASP, R-$\exists$ASP, $\forall$ASP, R-$\forall$ASP – one for each of the four combinations of types of attackers.
- We propose solutions for $\exists$ASP and R-$\exists$ASP via reduction to model-checking. The key idea of our approach is to replace the vulnerable processes - the victim(s) - by appropriate "gadgets", then ask a model-checker whether the resulting system violates a certain property.
- We implement our solutions in a prototype open-source tool called KORG, and apply KORG to TCP connection establishment and tear-down routines. Our experiments show KORG is able to automatically synthesize realistic, well-known attacks against TCP within seconds or minutes.

The rest of the paper is organized as follows. We present background material in §2. We define attacker synthesis problems in §3 and present solutions in §4. We describe the TCP case study in §5, present related work in §6, and conclude in §7.

2 Formal Model Preliminaries

We model distributed protocols as interacting processes, in the spirit of [1]. We next define formally these processes and their composition. We use 2^X to denote the power-set of X, and ω-exponentiation to denote infinite repetition, e.g., $a^\omega = aaa\cdots$.

2.1 Processes

Definition 1 (Process). *A* process *is a tuple* $P = \langle AP, I, O, S, s_0, T, L\rangle$ *with set of* atomic propositions *AP, set of* inputs I*, set of* outputs O*, set of* states S*,* initial state $s_0 \in S$*,* transition relation $T \subseteq S \times (I \cup O) \times S$*, and (total)* labeling function $L : S \to 2^{AP}$*, such that: AP,* I, O*, and* S *are finite; and* $I \cap O = \emptyset$*.*

Let $P = \langle \text{AP}, I, O, S, s_0, T, L\rangle$ be a process. For each state $s \in S$, $L(s)$ is a subset of AP containing the atomic propositions that are true at state s. Consider a transition (s, x, s') starting at state s and ending at state s' with label x. If the label x is an input, then the transition is called an *input transition* and denoted $s \xrightarrow{x?} s'$. Otherwise, x is an output, and the transition is called an *output transition* and denoted $s \xrightarrow{x!} s'$. A transition (s, x, s') is called *outgoing* from state s and *incoming* to state s'.

A state $s \in S$ is called a *deadlock* iff it has no outgoing transitions. The state s is called *input-enabled* iff, for all inputs $x \in I$, there exists some state $s' \in S$ such that there exists a transition $(s, x, s') \in T$. We call s an *input* state (or *output* sate) if all its outgoing transitions are input transitions (or output transitions, respectively).

A process P is *deterministic* iff all of the following hold: (i) its transition relation T can be expressed as a function $S \times (I \cup O) \to S$; (ii) every non-deadlock state in S is either an input state or an output state, but not both; (iii) input states are input-enabled; and (iv) each output state has only one outgoing transition. Determinism guarantees that: each state is a deadlock, an input state, or an output state; when a process outputs, its output is uniquely determined by its state; and when a process inputs, the input and state uniquely determine where the process transitions.

A *run* of a process P is an infinite sequence $r = \big((s_i, x_i, s_{i+1})\big)_{i=0}^{\infty} \subseteq T^{\omega}$ of consecutive transitions. We use runs(P) to denote all the runs of P. A run over states $s_0, s_1, ...$ induces a sequence of labels $L(s_0), L(s_1), ...$ called a *computation*.

2.2 Composition

The composition of two processes P_1 and P_2 is another process denoted $P_1 \parallel P_2$, capturing both the individual behaviors of P_1 and P_2 as well as their interactions with one another. We define the asynchronous parallel composition operator $\parallel$ with rendezvous communication as in [1].

Definition 2 (Process Composition). *Let $P_i = \langle AP_i, I_i, O_i, S_i, s_0^i, T_i, L_i \rangle$ be processes, for $i = 1, 2$. For the composition of P_1 and P_2 (denoted $P_1 \parallel P_2$) to be well-defined, the processes must have no common outputs, and no common atomic propositions. Then $P_1 \parallel P_2$ is defined below:*

$$P_1 \parallel P_2 = \langle AP_1 \cup AP_2, (I_1 \cup I_2) \setminus (O_1 \cup O_2), O_1 \cup O_2, S_1 \times S_2, (s_0^1, s_0^2), T, L \rangle \quad (1)$$

... where the transition relation T is precisely the set of transitions $(s_1, s_2) \xrightarrow{x} (s_1', s_2')$ such that, for $i = 1, 2$, if the label $x \in I_i \cup O_i$ is a label of P_i, then $s_i \xrightarrow{x} s_i' \in T_i$, else $s_i = s_i'$. $L : S_1 \times S_2 \to 2^{AP_1 \cup AP_2}$ is the function defined as $L(s_1, s_2) = L_1(s_1) \cup L_2(s_2)$.

The labeling function L is total as L_1 and L_2 are total. Since we required the processes P_1, P_2 to have disjoint sets of atomic propositions, L does not change the logic of the two processes under composition. Note that the composition of two processes is a process. Additionally, $\parallel$ is commutative and associative [1].

2.3 LTL

LTL [28] is a linear temporal logic for reasoning about computations. In this work, we use LTL to formulate properties of processes. The syntax of LTL is defined by the following grammar: $\phi ::= p \mid q \mid ... \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid \mathbf{X}\phi_1 \mid \phi_1 \mathbf{U} \phi_2$, where the $p \mid q \mid ...$ are any atomic propositions $\in$ AP, and ϕ_1, ϕ_2 can be any LTL formulae.

Let σ be a computation. If σ satisfies an LTL formula ϕ we write $\sigma \models \phi$. If $\neg(\sigma \models \phi)$, then we write $\sigma \not\models \phi$. The satisfaction relation for LTL is formally defined as follows: $\sigma \models p$ if p is true in $\sigma(0)$; $\sigma \models \mathbf{X}p$ if p is true in $\sigma(1)$; $\sigma \models \mathbf{F}p$ if there exists some $K \geq 0$ such that p is true in $\sigma(K)$; $\sigma \models \mathbf{G}p$ if for all $K \geq 0$, p is true in $\sigma(K)$; $\sigma \models p\mathbf{U}q$ if there exists some $K \geq 0$ such that for all $k_1 < K \leq k_2$, p is true in $\sigma(k_1)$ and q is true in $\sigma(q_2)$; and $\sigma \models \phi_1 \wedge \phi_2$ if $\sigma \models \phi_1$ and $\sigma \models \phi_2$.

An LTL formula ϕ is called a *safety property* iff it can be violated by a finite prefix of a computation, or a *liveness property* iff it can only be violated by an infinite computation [2]. For a process P and LTL formula ϕ, we write $P \models \phi$ iff, for every computation σ of P, $\sigma \models \phi$. For convenience, we naturally elevate our notation for satisfaction on computations to satisfaction on runs.

3 Attacker Synthesis Problems

We want to synthesize attackers automatically. Intuitively, an attacker is a process that, when composed with the system, violates some property. There are different types of attackers, depending on what it means to violate a property (in some cases? in all cases?), as well as on the system topology (threat model). Next, we define the threat model and attacker concepts formally, followed by the problems considered in this paper.

3.1 Threat Models

A *threat model* or *attacker model* prosaically captures the goals and capabilities of an attacker with respect to some victim and environment. Our threat model captures: how many attacker components there are; how they communicate with each other and with the rest of the system; and the attacker goals.

Definition 3 (Input-Output Interface). *An* input-output interface *is a tuple* (I, O) *such that* $I \cap O = \emptyset$ *and* $I \cup O \neq \emptyset$*. The* class *of an input-output interface* (I, O)*, denoted* $\mathcal{C}(I, O)$*, is the set of processes with inputs* I *and outputs* O*. Likewise,* $\mathcal{C}(P)$ *denotes the input-output interface the process* P *belongs to. (e.g. Fig. 2)*

Definition 4 (Threat Model). *A* threat model *is a tuple* $(P, (Q_i)_{i=0}^m, \phi)$ *where* $P, Q_0, ..., Q_m$ *are processes, each process* Q_i *has no atomic propositions (i.e., its set of atomic propositions is empty), and* ϕ *is an LTL formula such that* $P \parallel Q_0 \parallel ... \parallel Q_m \models \phi$*. We also require that the system* $P \parallel Q_0 \parallel ... \parallel Q_m$ *satisfies the formula* ϕ *in a non-trivial manner, that is, that* $P \parallel Q_0 \parallel ... \parallel Q_m$ *has at least one infinite run.*

In a threat model, the process P is called the *target process*, and the processes Q_i are called *vulnerable processes*. The goal of the adversary is to modify the vulnerable processes Q_i so that composition with the target process violates ϕ. (We assume that prior to the attack, the protocol behaves correctly, i.e., it satisfies ϕ.) See Fig. 1.

3.2 Attackers

Definition 5 (Attacker). *Let* TM $= (P, (Q_i)_{i=0}^m, \phi)$ *be a threat model. Then* $\boldsymbol{A} = (A_i)_{i=0}^m$ *is called a* TM-attacker *if* $P \parallel A_0 \parallel ... \parallel A_m \not\models \phi$*, and, for all* $0 \leq i \leq m$*:* A_i *is a deterministic process;* A_i *has no atomic propositions, and* $A_i \in \mathcal{C}(Q_i)$*.*

The existence of a $(P, (Q_i)_{i=0}^m, \phi)$-attacker means that if an adversary can exploit all the Q_i, then the adversary can attack P with respect to ϕ. Note that an attacker $\boldsymbol{A}$ cannot succeed by blocking the system from having any runs at all. Indeed, $P \parallel A_0 \parallel ... \parallel A_m \not\models \phi$ implies that $P \parallel A_0 \parallel ... \parallel A_m$ has at least one infinite run violating ϕ.

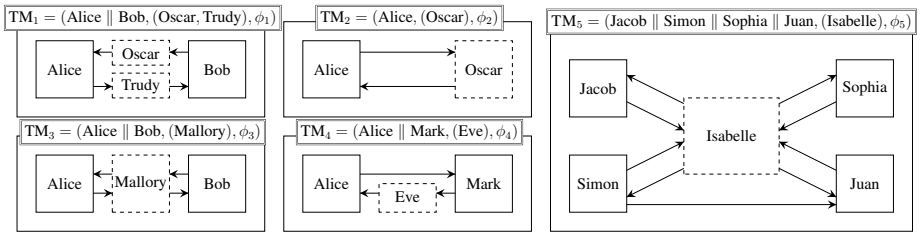


Fig. 1: Example Threat Models. The properties ϕ_i are not shown. Solid and dashed boxes are processes; we only assume the adversary can exploit the processes in the dashed boxes. TM_1 describes a distributed on-path attacker scenario, TM_2 describes an off-path attacker, TM_3 is a classical man-in-the-middle scenario, and TM_4 describes a one-directional man-in-the middle, or, depending on the problem formulation, an eavesdropper. TM_5 is a threat model with a distributed victim where the attacker cannot affect or read messages from Simon to Juan. Note that a directed edge in a network topology from Node 1 to Node 2 is logically equivalent to the statement that a portion of the outputs of Node 1 are also inputs to Node 2. In cases where the same packet might be sent to multiple recipients, the sender and recipient can be encoded in a message subscript. Therefore, the entire network topology is *implicit* in the interfaces of the processes in the threat model according to the composition definition.

Real-world computer programs implemented in languages like C or JAVA are called *concrete*, while logical models of those programs implemented as algebraic transition systems such as processes are called *abstract*. The motivation for synthesizing abstract attackers is ultimately to recover exploitation strategies that actually work against concrete protocols. So, we should be able to translate an abstract attacker (Fig. 2) into a concrete one (Fig. 8). Determinism guarantees that we can do this. We also require the attacker and the vulnerable processes to have no atomic propositions, so the attacker cannot "cheat" by directly changing the truth-hood of the property it aims to violate.

For a given threat model many attackers may exist. We want to differentiate attacks that are more effective from attacks that are less effective. One straightforward comparison is to partition attackers into those that always violate ϕ, and those that only sometimes violate ϕ. We formalize this notion with $\exists$-attackers and $\forall$-attackers.

Definition 6 ($\exists$-Attacker vs $\forall$-Attacker). *Let $\boldsymbol{A}$ be a $(P, (Q_i)_{i=0}^m, \phi)$-attacker. Then $\boldsymbol{A}$ is a* $\forall$-attacker *if $P \parallel A_0 \parallel ... \parallel A_m \models \neg\phi$. Otherwise, $\boldsymbol{A}$ is an* $\exists$-attacker.

A $\forall$-attacker $\boldsymbol{A}$ always succeeds, because $P \parallel \boldsymbol{A} \models \neg\phi$ means that *every* behavior of $P \parallel \boldsymbol{A}$ satisfies $\neg\phi$, that is, *every* behavior of $P \parallel \boldsymbol{A}$ violates ϕ. Since $P \parallel \boldsymbol{A} \not\models \phi$, there must exist a computation σ of $P \parallel \boldsymbol{A}$ such that $\sigma \models \neg\phi$, so, a $\forall$-attacker cannot succeed by blocking. An $\exists$-attacker is any attacker that is not a $\forall$-attacker, and every attacker succeeds in at least one computation, so an $\exists$-attacker sometimes succeeds, and sometimes does not. In most real-world systems, infinite attacks are impossible, implausible, or just uninteresting. To avoid such attacks, we define an attacker that produces finite-length sequences of adversarial behavior, and then "recovers", meaning that it behaves like the vulnerable process it replaced (see Fig. 3).

Definition 7 (Attacker with Recovery). *Let $\boldsymbol{A}$ be a $(P, (Q_i)_{i=0}^m, \phi)$-attacker. If, for each $0 \leq i \leq m$, the attacker component A_i consists of a finite directed acyclic graph*

(DAG) ending in the initial state of the vulnerable process Q_i, followed by all of the vulnerable process Q_i, then we say the attacker $\boldsymbol{A}$ is an attacker with recovery. *We refer to the Q_i postfix of each A_i as its* recovery .

Note that researchers sometimes use "recovery" to mean when a system undoes the damage caused by an attack. We use the word differently, to mean when the property ϕ remains violated even under *modus operandi* subsequent to attack termination.

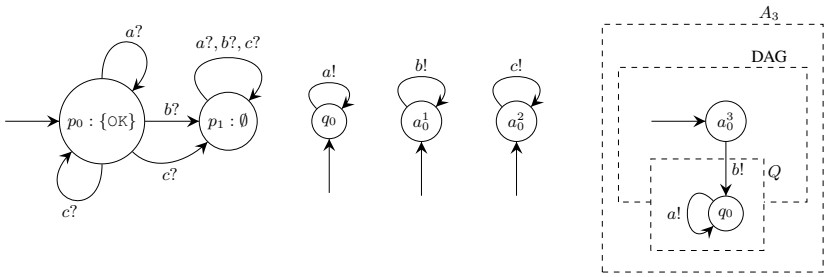


Fig. 2: From left to right: processes P, Q, A_1, A_2, A_3. Let $\phi = \mathbf{G}\,\texttt{OK}$, and let the interface of Q be $\mathcal{C}(Q) = (\emptyset, \{a, b, c\})$. Then $P \parallel Q \models \phi$. A_1 and A_2 are both deterministic and have no input states. Let $\mathcal{C}(A_1) = \mathcal{C}(A_2) = \mathcal{C}(Q)$. Then, A_1 and A_2 are both $(P, (Q), \phi)$-attackers. A_1 is a $\forall$-attacker, and A_2 is an $\exists$-attacker. A_3 is a $\forall$-attacker with recovery consisting of a DAG starting at a_0^3 and ending at the initial state q_0 of Q, plus all of Q, namely the recovery.

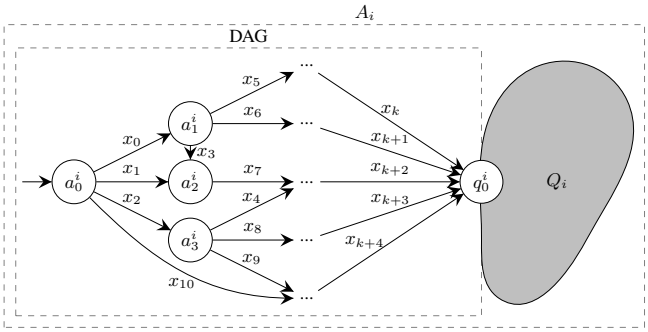


Fig. 3: Suppose $\boldsymbol{A} = (A_i)_{i=0}^m$ is attacker with recovery for TM $= (P, (A_i)_{i=0}^m, \phi)$. Further suppose A_i has initial state a_0^i, and Q_i has initial state q_0^i. Then A_i should consist of a DAG starting at a_0^i and ending at q_0^i, plus all of Q_i, called the *recovery*, indicated by the shaded blob. Note that if some Q_i is non-deterministic, then there can be no attacker with recovery, because Q_i is a subprocess of A_i, and all the A_is must be deterministic in order for $\boldsymbol{A}$ to be an attacker.

3.3 Attacker Synthesis Problems

Each type of attacker - $\exists$ versus $\forall$, with recovery versus without - naturally induces a synthesis problem.

Problem 1 ($\exists$-Attacker Synthesis Problem ($\exists$ASP)). Given a threat model TM, find a TM-attacker, if one exists; otherwise state that none exists.

Problem 2 (Recovery ∃-Attacker Synthesis Problem (R-∃ASP)). Given a threat model TM, find a TM-attacker with recovery, if one exists; otherwise state that none exists.

We defined ∃ and ∀-attackers to be disjoint, but, if the goal is to find an ∃-attacker, then surely a ∀-attacker is acceptable too; we therefore did not restrict the ∃-problems to only ∃-attackers. Next we define the two ∀-problems, which remain for future work.

Problem 3 (∀-Attacker Synthesis Problem (∀ASP)). Given a threat model TM, find a TM-∀-attacker, if one exists; otherwise state that none exists.

Problem 4 (Recovery ∀-Attacker Synthesis Problem (R-∀ASP)). Given a threat model TM, find a TM-∀-attacker with recovery, if one exists; otherwise state that none exists.

4 Solutions

We present solutions ∃ASP and R-∃ASP for any number of attackers, and for both safety and liveness properties. Our success criteria are soundness and completeness. Both solutions are polynomial in the product of the size of P and the sizes of the interfaces of the Q_is, and exponential in the size of the property ϕ [35]. For real-world performance, see Section 5.

We reduce ∃ASP and R-∃ASP to model-checking. The idea is to replace the vulnerable processes Q_i with appropriate "gadgets", then ask a model-checker whether the system violates a certain property. We prove that existence of a violation (a counterexample) is equivalent to existence of an attacker, and we show how to transform the counterexample into an attacker. The gadgets and the LTL formula are different, depending on whether we seek attackers without or with recovery.

4.1 Gadgetry

A computation σ is a *lasso* if it equals a finite word α, then infinite repetition of a finite word β, i.e., $\sigma = \alpha \cdot \beta^\omega$. A prefix α of a computation σ is called a *bad prefix* for P and ϕ if P has ≥ 1 runs inducing computations starting with α, and every computation starting with α violates ϕ. We naturally elevate the terms *lasso* and *bad prefix* to runs and their prefixes. We assume a *model checker*: a procedure $\text{MC}(P, \phi)$ that takes as input a process P and property ϕ, and returns $\emptyset$ if $P \models \phi$, or one or more violating lasso runs or bad prefixes of runs for P and ϕ, otherwise [2].

Attackers cannot have atomic propositions. So, the only way for $\boldsymbol{A}$ to *attack* TM is by sending and receiving messages, hence the space of attacks is within the space of labeled transition sequences. The *Daisy Process* nondeterministically exhausts the space of input and output events of a vulnerable process.

Definition 8 (Daisy Process). *Let $Q = \langle \emptyset, I, O, S, s_0, T, L \rangle$ be a process with no atomic propositions. Then the* daisy *of Q, denoted* $\textsc{Daisy}(Q)$*, is the process defined below, where $L' : \{d_0\} \to \{\emptyset\}$ is the map such that $L'(d_0) = \emptyset$.*

$$\textsc{Daisy}(Q) = \langle \emptyset, I, O, \{d_0\}, d_0, \{(d_0, w, d_0) \mid w \in I \cup O\}, L' \rangle \tag{2}$$

Next, we define a *Daisy with Recovery*. This gadget is an *abstract process*, i.e., a generalized process with a non-empty set of initial states $S_0 \subseteq S$. Composition and LTL semantics for abstract processes are naturally defined. We implicitly transform processes to abstract processes by wrapping the initial state in a set.

Definition 9 (Daisy with Recovery). *Given a process $Q_i = \langle \emptyset, I, O, S, s_0, T, L \rangle$, the* daisy with recovery *of Q_i, denoted* $\text{RDAISY}(Q_i)$*, is the abstract process* $\text{RDAISY}(Q_i) = \langle AP, I, O, S', S_0, T', L' \rangle$*, with atomic propositions $AP = \{\texttt{recover}_i\}$, states $S' = S \cup \{d_0\}$, initial states $S_0 = \{s_0, d_0\}$, transitions $T' = T \cup \{(d_0, x, w_0) \mid x \in I \cup O, w_0 \in S_0\}$, and labeling function $L' : S' \to 2^{AP}$ that takes s_0 to $\{\texttt{recover}_i\}$ and other states to $\emptyset$. (We reserve the symbols $\texttt{recover}_0, ...$ for use in daisies with recovery, so they cannot be sub-formulae of the property in any threat model.)*

4.2 Solution to ∃ASP

Let $\text{TM} = (P, (Q_i)_{i=0}^m, \phi)$ be a threat model. Our goal is to find an attacker for TM, if one exists, or state that none exists, otherwise. First, we check whether the system $P \parallel \text{DAISY}(Q_0) \parallel ... \parallel \text{DAISY}(Q_m)$ satisfies ϕ. If it does, then no attacker exists, as the daisy processes encompass any possible attacker behavior. Define a set R returned by the model-checker MC:

$$R = \text{MC}(P \parallel \text{DAISY}(Q_0) \parallel ... \parallel \text{DAISY}(Q_m), \phi) \tag{3}$$

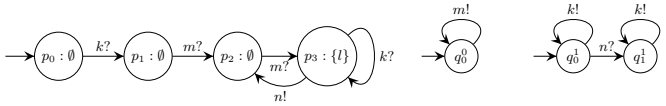
If $R = \emptyset$ then no attacker exists. On the other hand, if the system violates ϕ, then we can transform a violating run into a set of attacker processes by projecting it onto the corresponding interfaces. Choose a violating run or bad prefix $r \in R$ arbitrarily. Either $r = \alpha$ is some finite bad prefix, or $r = \alpha \cdot \beta^\omega$ is a violating lasso. For each $0 \leq i \leq m$, let α_i be the projection of α onto the process $\text{DAISY}(Q_i)$. That is, let $\alpha_i = []$; then for each $(\boldsymbol{s}, x, \boldsymbol{s'})$ in α, if x is an input or an output of Q_i, and q, q' are the states $\text{DAISY}(Q_i)$ embodies in $\boldsymbol{s}, \boldsymbol{s'}$, add (q, x, q') to α_i. For each α_i, create an incomplete process A_i^α with a new state s_{j+1}^α and transition $s_j^\alpha \xrightarrow{z} s_{j+1}^\alpha$ for each $\alpha_i[j] = (d_0^i, z, d_0^i)$ for $0 \leq j < |\alpha_i|$. If $r = \alpha \cdot \beta^\omega$ is a lasso, then for each $0 \leq i \leq m$, define A_i^β from β_i in the same way that we defined A_i^α from α_i; let A_i' be the result of merging the first and last states of A_i^β with the last state of A_i^α. Otherwise, if $r = \alpha$ is a bad prefix, let A_i' be the result of adding an input self-loop to the last state of A_i^α, or an output self-loop if Q_i has no inputs. Either way, A_i' is an incomplete attacker. Finally let A_i be the result of making every input state in A_i' input-enabled via self-loops, and return the attacker $\boldsymbol{A} = (A_i)_{i=0}^m$. An illustration of the method is given in Figure 4.

Theorem 1 (∃ASP Solution is Sound and Complete). *Let* $\text{TM} = (P, (Q_i)_{i=0}^m, \phi)$ *be a threat model, and define R as in Eqn. 3. Then the following hold. 1) $R \neq \emptyset$ iff a* TM*-attacker exists. 2) If $R \neq \emptyset$, then the procedure above eventually returns a* TM*-attacker.*

4.3 Solution to R-∃ASP

Let $\text{TM} = (P, (Q_i)_{i=0}^m, \phi)$ be a threat model as before. Now our goal is to find a TM-attacker *with recovery*, if one exists, or state that none exists, otherwise. The idea to

Threat Model: $\text{TM}' = (P, (Q_0, Q_1), \phi)$, where the processes from left to right are P, Q_0, and Q_1, and where $\phi = \mathbf{FG}\, l$. P has inputs k, m, and output n. Q_0 has no inputs, and output m. Q_1 has inputs n, h, and output k. Recall that $P \parallel Q_0 \parallel Q_1 \models \phi$.



Violating run: A run $r \in R$ where R is defined as in Equation 3.

$$r = \overbrace{\begin{bmatrix} p_0 \\ d_0^0 \\ d_0^1 \end{bmatrix} \xrightarrow{k!} \begin{bmatrix} p_1 \\ d_0^0 \\ d_0^1 \end{bmatrix} \xrightarrow{m!}}^{\alpha} \overbrace{\begin{bmatrix} p_2 \\ d_0^0 \\ d_0^1 \end{bmatrix} \xrightarrow{m!} \begin{bmatrix} p_3 \\ d_0^0 \\ d_0^1 \end{bmatrix} \xrightarrow{n!} \begin{bmatrix} p_2 \\ d_0^0 \\ d_0^1 \end{bmatrix} \xrightarrow{m!} \begin{bmatrix} p_3 \\ d_0^0 \\ d_0^1 \end{bmatrix} \xrightarrow{n!} \ldots}^{\beta^\omega} \in R$$

Application of solution: r is projected and translated into an attacker $\boldsymbol{A} = (A_0, A_1)$.

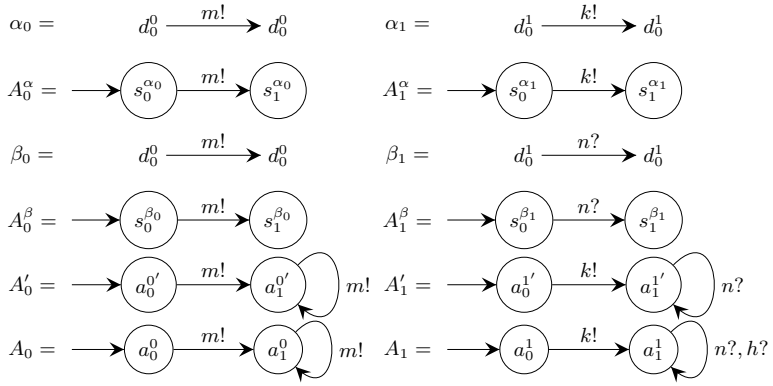


Fig. 4: Example threat model TM' on top, followed by a violating run in R, followed by translation of the run into attacker.

solve this problem is similar to the idea for finding attackers without recovery, with two differences. First, the daisy processes are now more complicated, and include recovery to the original Q_i processes. Second, the formula used in model-checking is not ϕ, but a more complex formula ψ to ensure that the attacker eventually recovers, i.e., all the attacker components eventually recover. We define the property ψ so that in prose it says "if all daisies eventually recover, then ϕ holds". We then define R like before, except we replace daisies with daisies with recovery, and ϕ with ψ, as defined below.

$$\psi = \Big(\bigwedge_{0 \le i \le m} \mathbf{F}\, \texttt{recover}_i \Big) \implies \phi \tag{4}$$

$$R = \text{MC}(P \parallel \text{RDAISY}(Q_0) \parallel \ldots \parallel \text{RDAISY}(Q_m), \psi) \tag{5}$$

If $R = \emptyset$ then no attacker with recovery exists. If any Q_i is not deterministic, then likewise no attacker with recovery exists, because our attacker definition requires the attacker to be deterministic, but if Q_i is not and $Q_i \subseteq A_i$ then neither is A_i.

Otherwise, choose a violating run (or bad prefix) $r \in R$ arbitrarily. We proceed as we did for ∃ASP but with three key differences. First, we define α_i by projecting α onto $\text{RDAISY}(Q_i)$ as opposed to $\text{DAISY}(Q_i)$. Second, for each $0 \leq i \leq m$, instead of using A_i^β if r is a lasso, or adding self-loops to the final state if r is a bad prefix, we simply glue A_i^α to Q_i by setting the last state of A_i^α to be the initial state of Q_i. (The result of gluing is a process; the initial state of A_i^α is its only initial state.) Third, instead of using self-loops to input-enable input states, we use input transitions to the initial state of Q_i. This ensures the pre-recovery portion is a DAG. Then we return $\boldsymbol{A} = (A_i)_{i=0}^m$.

Theorem 2 (R-∃ASP Solution is Sound and Complete). *Let* $\text{TM} = (P, (Q_i)_{i=0}^m, \phi)$ *be a threat model, and define* R *as in Eqn. 5. Assume all the* Q_i*s are deterministic. Then the following hold. 1)* $R \neq \emptyset$ *iff a* TM*-attacker with recovery exists. 2) If* $R \neq \emptyset$*, then the procedure described above eventually returns a* TM*-attacker with recovery.*

5 Case Study: TCP

Implementation We implemented our solutions in an open-source tool called KORG. We say an attacker $\boldsymbol{A}$ for a threat model $\text{TM} = (P, (Q_i)_{i=0}^m, \phi)$ is a *centralized attacker* if $m = 0$, or a *distributed attacker*, otherwise. In other words, a centralized attacker has only one attacker component $\boldsymbol{A} = (A)$, whereas a distributed attacker has many attacker components $\boldsymbol{A} = (A_i)_{i=0}^m$. KORG handles ∃ASP and R-∃ASP for liveness and safety properties for a centralized attacker. KORG is implemented in PYTHON 3 and uses the model-checker SPIN [15] as its underlying verification engine.

TCP is a fundamental Internet protocol consisting of three stages: connection establishment, data transfer, and connection tear-down. We focus on the first and third stages, which jointly we call the connection routine. Our approach and model (see Fig. 5, 6) are inspired by SNAKE [19]. Run-times and results are listed in Table 1.

Fig. 5: TCP threat model block diagram. Each box is a process. An arrow from process P_1 to process P_2 denotes that a subset of the outputs of P_2 are exclusively inputs of P_1. PEERs 1 and 2 are TCP peers. A *channel* is a directed FIFO queue of size one with the ability to detect fullness. A full channel may be overwritten. 1TON, NTO1, 2TON, and NTO2 are channels. Implicitly, channels relabel: for instance, 1TON relabels outputs from PEER 1 to become inputs of NETWORK; NETWORK transfers messages between peers via channels, and is the vulnerable process.

Threat Models Rather than communicating directly with the NETWORK, the peers communicate with the channels, and the channels communicate with the NETWORK, allowing us to model the fact that packets are not instantaneously transferred in the wild. We use the shorthand CHAN!MSG to denote the event where MSG is sent over a channel CHAN; it is contextually clear who sent or received the message. We abstract the lower network stack layer TCP relies on with NETWORK, which passes messages between 1TON ∥ 2TON and NTO1 ∥ NTO2. We model the peers symmetrically.

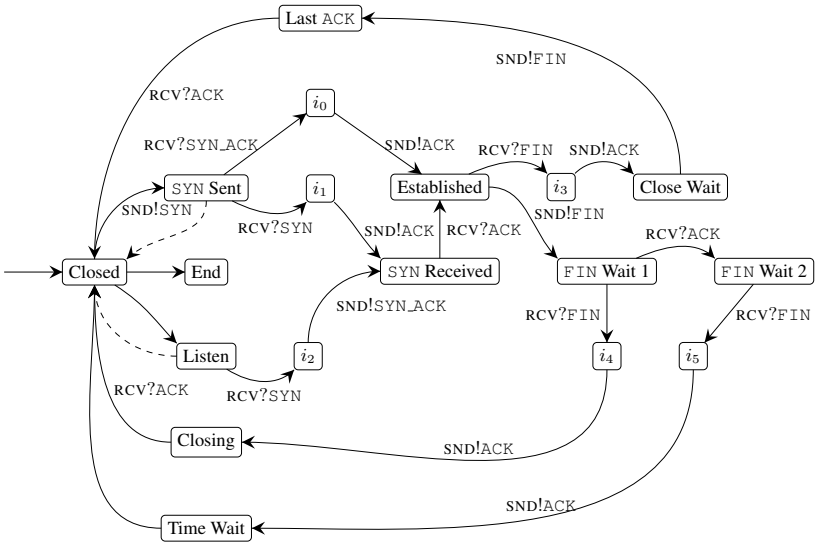


Fig. 6: A TCP peer. For $i = 1, 2$, if this is PEER i, then SND $:=$ iTON and RCV $:=$ NTOi. All the states except $i_0, ..., i_5$, and End are from the finite state machine in the TCP RFC [29]. The RFC diagram omits the implicit states $i_0, ..., i_5$, instead combining send and receive events on individual transitions. In the RFC, Closed is called a "fictional state", where no TCP exists. We add a state End to capture the difference between a machine that elects not to instantiate a peer and a machine that is turned off. We label each state s with a single atomic proposition s_i. Dashed transitions are *timeout transitions*, meaning they are taken when the rest of the system deadlocks.

Given a property ϕ about TCP, we can formulate a threat model TM as follows, where we assume the adversary can exploit the lower layers of a network and ask if the adversary can induce TCP to violate ϕ:

$$\text{TM} = (\text{PEER 1} \parallel \text{PEER 2} \parallel \text{1TON} \parallel \text{2TON} \parallel \text{NTO1} \parallel \text{NTO2}, (\text{NETWORK}), \phi) \quad (6)$$

We consider the properties ϕ_1, ϕ_2, ϕ_3, giving rise to the threat models $\text{TM}_1, \text{TM}_2, \text{TM}_3$.

TM_1: No Half-Closed Connection Establishment The safety property ϕ_1 says that if PEER 1 is in Closed state, then PEER 2 cannot be in Established state.

$$\phi_1 = \mathbf{G}(\text{Closed}_1 \implies \neg\text{Established}_2) \quad (7)$$

KORG discovers an attacker that spoofs the active participant in an active-passive connection establishment (see message sequence chart in Fig. 7), as described in [13].

TM_2: Passive-Active Connection Establishment Eventually Succeeds The liveness property ϕ_2 says that if it is infinitely often true that PEER 1 is in Listen state while PEER 2 is in SYN Sent state, then it must eventually be true that PEER 1 is in Established state.

$$\phi_2 = (\mathbf{GF}(\text{Listen}_1 \wedge \text{SYN Sent}_2)) \implies \mathbf{F}\,\text{Established}_1 \quad (8)$$

KORG discovers an attack where a SYN packet from PEER 2 is dropped. The corresponding attacker code is given in the PROMELA language of SPIN in Fig. 8.

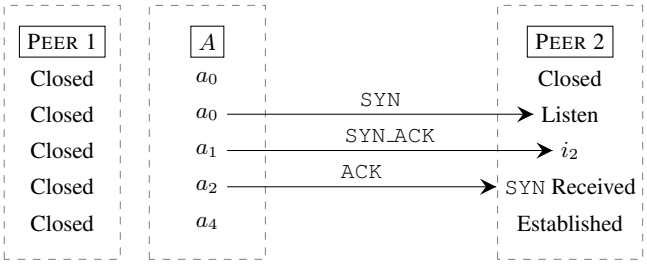


Fig. 7: Time progresses from top to bottom. Labeled arrows denote message exchanges over implicit channels. The property is violated in the final row; after this recovery may begin.

```
Nto1 ! ACK; Nto2 ! ACK; 2toN ? SYN; /* ... recovery ... */
```

Fig. 8: Body of PROMELA process for a TM_2-attacker with recovery generated by KORG. PEER 2 transitions from Closed state to SYN Sent state and sends SYN to PEER 1. The attacker drops this packet so that it never reaches PEER 1. PEER 1 then transitions back and forth forever between Closed and Listen states, and the property is violated. Because SPIN attempts to find counterexamples as quickly as possible, the counterexamples it produces are not in general minimal.

TM_3: Peers Do Not Get Stuck The safety property ϕ_3 says that the two peers will never simultaneously deadlock outside their End states. Let S_i denote the set of states in Fig. 6 for PEER i, and $S_i' = S_i \setminus \{\text{End}\}$.

$$\phi_3 = \bigwedge_{s_1 \in S_1'} \bigwedge_{s_2 \in S_2'} \neg \mathbf{FG}(s_1 \wedge s_2) \tag{9}$$

For the problem with recovery, KORG discovers an attacker that selectively drops the ACK sent by PEER 1 as it transitions from i_0 to Established state in an active/passive connection establishment routine, leaving PEER 2 stranded in SYN Received state, leading to a violation of ϕ_3. Similar bugs exist in real-world implementations, e.g. [31].

Performance Performance results for Case Study are given in Table 1. Our success criteria was to produce realistic attackers faster than an expert human could with pen-and-paper. We discovered attackers in seconds or minutes as shown in Table 1.

6 Related Work

Prior works formalized security problems using game theory (e.g., FLIPIT [34], [22]), "weird machines" [7], attack trees [37], Markov models [33], and other methods. Prior notions of attacker quality include $\mathcal{O}$-complexity [6], expected information loss [30], or success probability [25, 36], which is similar to our concept of $\forall$ versus $\exists$-attackers. The formalism of [36] also captures attack consequence (cost to a stakeholder).

Attacker synthesis work exists in cyber-physical systems [27, 3, 18, 25], most of which define attacker success using bad states (e.g., reactor meltdown, vehicle collision, etc.) or information theory (e.g., information leakage metrics). Problems include the *actuator attacker synthesis problem* [23]; the *hardware-aware attacker synthesis problem* [32]; and the *fault-attacker synthesis problem* [4].

Property	Avg. Runtime (s) Unique Attacker		Unique Attackers Found	
	∃ASP	R-∃ASP	∃ASP	R-∃ASP
ϕ_1	0.32	0.49	7	5
ϕ_2	0.45	0.48	5	5
ϕ_3	876.74	2757.98	4	5

Table 1: For each property ϕ_i, we asked KORG 10 times to generate 10 attackers with recovery, and 10 without, on a 16Gb 2018 Intel© Core$^{\text{tm}}$ i7-8550U CPU running Linux Mint 19.3 Cinnamon. KORG may generate duplicate attackers, so for each property (Column 1), we list the average time to generate a unique attacker without recovery (Column 2) or with (Column 3), and the total number of unique attackers found without recovery (Column 4) or with (Column 5). E.g., for ϕ_3, of 100 attackers with recovery generated over about four hours, five were unique and 95 duplicates, so KORG took about 2.3 minutes per attacker, or, 45 minutes per unique attacker. Instructions and code to reproduce these results are given in the GitHub repository.

Maybe the most similar work to our own is PROVERIF [5], which verifies properties of, and generates attacks against, cryptographic protocols. We formalize the problem with operational semantics (processes) and reduce it to model checking, whereas PROVERIF uses axiomatic semantics (PROLOG clauses) and reduces it to automated proving. Another similar tool is NETSMC [39], a model-checker that efficiently finds counter-examples to security properties of stateful networks.

Existing techniques for automated attack discovery include model-guided search [19, 16] (including using inference [8]), open-source-intelligence [38], bug analysis [17], and genetic programming [21]. The generation of a failing test-case for a protocol property is not unlike attack discovery, so [24] is also related.

This paper focuses on attacker synthesis at the protocol level, and thus differs from the work reported in [20] in two ways: first, the work in [20] synthesizes mappings between high-level protocol models and execution platform models, thereby focusing on linking protocol design and implementation; second, the work in [20] synthesizes correct (secure) mappings, whereas we are interested in synthesizing attackers.

7 Conclusion

We present a novel formal framework for automated attacker synthesis. The framework includes an explicit definition of threat models and four novel, to our knowledge, categories of attackers. We formulate four attacker synthesis problems, and propose solutions to two of them by program transformations and reduction to model-checking. We prove our solutions sound and complete; these proofs are available online [14]. Finally, we implement our solutions for the case of a centralized attacker in an open-source tool called KORG, apply KORG to the study of the TCP connection routine, and discuss the results. KORG and the TCP case study are freely and openly available[1].

Acknowledgments This material is based upon work supported by the National Science Foundation under NSF SaTC award CNS-1801546. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors thank four anonymous reviewers. Additionally, the first author thanks Benjamin Quiring, Dr. Ming Li, and Dr. Frank von Hippel.

[1] `github.com/maxvonhippel/AttackerSynthesis`

References

1. Alur, R., Tripakis, S.: Automatic synthesis of distributed protocols. SIGACT News **48**(1), 55–90 (2017)
2. Baier, C., Katoen, J.P.: Principles of model checking. MIT press (2008)
3. Bang, L., Rosner, N., Bultan, T.: Online synthesis of adaptive side-channel attacks based on noisy observations. In: 2018 IEEE European Symposium on Security and Privacy. pp. 307–322. IEEE (2018)
4. Barthe, G., Dupressoir, F., Fouque, P.A., Grégoire, B., Zapalowicz, J.C.: Synthesis of fault attacks on cryptographic implementations. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 1016–1027 (2014)
5. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th IEEE Computer Security Foundations Workshop. pp. 82–96. IEEE Computer Society, Cape Breton, Nova Scotia, Canada (Jun 2001)
6. Branco, R., Hu, K., Kawakami, H., Sun, K.: A mathematical modeling of exploitations and mitigation techniques using set theory. In: 2018 IEEE Security and Privacy Workshops (SPW). pp. 323–328. IEEE (2018)
7. Bratus, S., Locasto, M.E., Patterson, M.L., Sassaman, L., Shubina, A.: Exploit programming: From buffer overflows to weird machines and theory of computation. USENIX; login **36**(6) (2011)
8. Cho, C.Y., Babic, D., Poosankam, P., Chen, K.Z., Wu, E.X., Song, D.: MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In: USENIX Security Symposium. vol. 139 (2011)
9. Chong, S., Guttman, J., Datta, A., Myers, A., Pierce, B., Schaumont, P., Sherwood, T., Zeldovich, N.: Report on the NSF workshop on formal methods for security. (2016)
10. Church, A.: Application of recursive arithmetic to the problem of circuit synthesis (7 1957), as cited in doi:10.2307/2271310.
11. Dijkstra, E.W., et al.: Notes on structured programming. `http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF` (1970), accessed: 11 May 2020
12. Duran, J.W., Ntafos, S.: A report on random testing. In: Proceedings of the 5th international conference on Software engineering. pp. 179–183. IEEE Press (1981)
13. Friedrichs, O.: A simple TCP spoofing attack. `citi.umich.edu/u/provos/papers/secnet-spoof.txt` (February 1997), accessed: 3 January 2020
14. von Hippel, M., Vick, C., Tripakis, S., Nita-Rotaru, C.: Automated attacker synthesis for distributed protocols. arXiv preprint arXiv:2004.01220 (2020)
15. Holzmann, G.: The Spin Model Checker. Addison-Wesley (2003)
16. Hoque, E., Chowdhury, O., Chau, S.Y., Nita-Rotaru, C., Li, N.: Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 627–638. IEEE (2017)
17. Huang, S.K., Huang, M.H., Huang, P.Y., Lai, C.W., Lu, H.L., Leong, W.M.: Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In: 2012 IEEE Sixth International Conference on Software Security and Reliability. pp. 78–87. IEEE (2012)
18. Huang, Z., Etigowni, S., Garcia, L., Mitra, S., Zonouz, S.: Algorithmic attack synthesis using hybrid dynamics of power grid critical infrastructures. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 151–162. IEEE (2018)
19. Jero, S., Lee, H., Nita-Rotaru, C.: Leveraging state information for automated attack discovery in transport protocol implementations. In: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. pp. 1–12. IEEE (2015)

20. Kang, E., Lafortune, S., Tripakis, S.: Automated Synthesis of Secure Platform Mappings. In: Computer Aided Verification (CAV) (Jul 2019)
21. Kayacik, H.G., Zincir-Heywood, A.N., Heywood, M.I., Burschka, S.: Generating mimicry attacks using genetic programming: a benchmarking study. In: 2009 IEEE Symposium on Computational Intelligence in Cyber Security. pp. 136–143. IEEE (2009)
22. Klaška, D., Kučera, A., Lamser, T., Řehák, V.: Automatic synthesis of efficient regular strategies in adversarial patrolling games. In: Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems. pp. 659–666. International Foundation for Autonomous Agents and Multiagent Systems (2018)
23. Lin, L., Zhu, Y., Su, R.: Synthesis of actuator attackers for free. arXiv preprint arXiv:1904.10159 (2019)
24. McMillan, K.L., Zuck, L.D.: Formal specification and testing of QUIC. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 227–240. ACM (2019)
25. Meira-Góes, R., Kwong, R., Lafortune, S.: Synthesis of sensor deception attacks for systems modeled as probabilistic automata. In: 2019 American Control Conference. pp. 5620–5626. IEEE (2019)
26. Myers, G.J.: The art of software testing. John Wiley & Sons (1979)
27. Phan, Q.S., Bang, L., Pasareanu, C.S., Malacaria, P., Bultan, T.: Synthesis of adaptive side-channel attacks. In: 2017 IEEE 30th Computer Security Foundations Symposium. pp. 328–342. IEEE (2017)
28. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science. pp. 46–57. IEEE (1977)
29. Postel, J., et al.: Rfc 793 Transmission Control Protocol (September 1981)
30. Srivastava, H., Dwivedi, K., Pankaj, P.K., Tewari, V.: A formal attack centric framework highlighting expected losses of an information security breach. International Journal of Computer Applications **68**(17) (2013)
31. `@henryouly`: [Solved] TCP connection blocked in SYN_SENT status. `bbs.archlinux.org/viewtopic.php?id=33875` (2007), accessed: 3 January 2020
32. Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro **39**(3), 84–93 (2019)
33. Valizadeh, S., van Dijk, M.: Toward a theory of cyber attacks. arXiv preprint arXiv:1901.01598 (2019)
34. Van Dijk, M., Juels, A., Oprea, A., Rivest, R.L.: FlipIt: The game of stealthy takeover. Journal of Cryptology **26**(4), 655–713 (2013)
35. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of the First Symposium on Logic in Computer Science. pp. 322–331. IEEE Computer Society (1986)
36. Vasilevskaya, M., Nadjm-Tehrani, S.: Quantifying risks to data assets using formal metrics in embedded system design. In: International Conference on Computer Safety, Reliability, and Security. pp. 347–361. Springer (2014)
37. Wideł, W., Audinot, M., Fila, B., Pinchinat, S.: Beyond 2014: Formal methods for attack tree–based security modeling. ACM Computing Surveys **52**(4), 1–36 (2019)
38. You, W., Zong, P., Chen, K., Wang, X., Liao, X., Bian, P., Liang, B.: Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2139–2154 (2017)
39. Yuan, Y., Moon, S.J., Uppal, S., Jia, L., Sekar, V.: NetSMC: A custom symbolic model checker for stateful network verification. In: 17th USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, Santa Clara, CA (Feb 2020)