Formal Methods in Computer-Aided Design 2023

CRV: Automated Cyber-Resiliency Reasoning for System Design Models

Daniel Larraz*, Robert Lorch*, Moosa Yahyazadeh*, M. Fareed Arif†,
Omar Chowdhury‡, Cesare Tinelli*
*The University of Iowa, Iowa City, USA, ✉ daniel-larraz@uiowa.edu
†The University of Oxford, Oxford, UK ‡Stony Brook University, Stony Brook, USA

***Abstract*—We present the design and implementation of an automated static analysis approach and corresponding diagnostic tool, called Cyber Resiliency Verifier (CRV), to check whether a system design satisfies its end-to-end guarantees when the integrity of one or more of its components cannot be guaranteed. CRV's key insight is to reason about *effects of integrity attacks* instead of concrete attacks, enabling it to reason also about the impact of future attacks having the same captured effects. We demonstrate CRV's effectiveness with a case study on a realistic design of an unmanned aerial delivery drone.**

I. Introduction

Security vulnerabilities in critical systems can have catastrophic impacts. Even when a vulnerability is discovered, performing root cause analysis and then adding security mechanisms *a posteriori* can be expensive, challenging, or infeasible. Exploitable weaknesses in a system's design are arguably harder to mitigate after deployment due to backward compatibility requirements, operational cost, and QoS constraints. This paper focuses on enabling system architects to identify and mitigate such design weaknesses *at the system design stage*.

A major reason systems have exploitable design weaknesses is that during the design phase, security considerations often take a secondary role to other requirements such as time to market. In addition, current design analysis tools and methodologies often pay scant attention to security considerations. The lack of sophisticated capabilities for the identification of security vulnerabilities at an abstract design level is an impediment for the model-based system design paradigm to reach its full potential. Although it is impossible to fully avoid vulnerabilities in the implementation, model-based analysis tools can nevertheless help designers design systems with *cyber-resiliency* in mind, that is, design systems whose functionality and integrity guarantees degrade gracefully under attack. We broadly define a system's *integrity properties/guarantees* as functional properties that must be satisfied for achieving its desired functionality. *We propose a general approach and a highly automated tool,* CRV, *whose rich diagnostic information allows a system architect to assess under different threat models the resiliency of a system design with respect to desired integrity properties*. A static analysis tool like CRV allows a system architect to account for security considerations already in the design phase. In particular, it enables *what-if*-type analyses exploring the effect that violations of integrity properties in a sub-system or software component may have on the overall system guarantees, avoiding surprises like a recent supply chain attack [29].

A typical workflow prescribed by CRV starts with the system architect developing a system model in a suitable modeling language and identifying critical functional properties that the system must maintain even when subject to attacks that compromise one or more system components. A system's design model (or, *system design*) contains *system architecture* information as well as *behavioral information* on one or more components. For cyber-resiliency analysis, the designer additionally selects a subset of the model's components and/or connections whose integrity cannot guaranteed. CRV then automatically instruments the design to account for the integrity issues. This instrumentation reduces the problem of assessing the cyber-resiliency of the design to a model checking problem: the satisfaction of the original functional properties also in the instrumented model. A violation of one of these properties implies that the original design (before instrumentation) is not resilient in the presence of attacks captured by the chosen threat model. When CRV discovers a violation, it provides as evidence an execution trace of the instrumented system for each violated property. These traces demonstrate the viability of attacks from the chosen threat model, as well as their effects on system properties. Additionally, for each violation, CRV provides a list of system components whose misbehavior may have contributed to the violation. For each property that remains satisfied even under attack, CRV provides a list of critical sub-components that may have contributed to satisfaction of the property. We demonstrate the effectiveness of CRV by evaluating it with respect to a case study of an UDD.

Contributions. In summary, this work makes the following technical contributions: (i) a *general framework and tool* for analyzing the resiliency of a system design with respect to desired functional properties against one or more threat models, including *replay attacks*; (ii) a novel notion of *attack/threat effects* that allows for the automatic instrumentation of design models and takes into account both known and unknown integrity attacks by reducing resiliency analysis to model checking; (iii) a formalization of attack effects in terms of the Dolev-Yao model, a formal model common in cryptographic protocol verification; (iv) a new automated process, *blame assignment*, to identify compromised components of a system

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_29

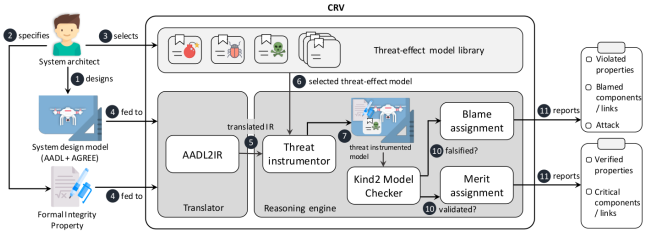

Fig. 1: CRV Architecture and Workflow Diagram.

design whose misbehavior can contribute to the violation of one or more functional properties of the system; (v) an automated process, *merit assignment*, that identifies components of a system design that positively impact the satisfaction of desired properties; and (vi) a *case study* of CRV against a model of an unmanned delivery drone.[1]

II. Design Overview of CRV

We start this section with the problem definition. We then present CRV's architecture, describe its interactions with the system designer, and discuss the underlying challenges.

A. Problem Definition and Scope

CRV automatically checks if a system design provides the required functional guarantees when the integrity of some of its sub-component cannot be ensured. CRV reasons about a hierarchical system design $\mathcal{D}$ specifying the system architecture, the function behavior of each component, one or more functional guarantees Φ_i, and a threat model $\mathcal{T}_m$ indicating which components and connections are vulnerable to attacks. CRV attempts to prove whether $\mathcal{D}$ can maintain Φ_i even when, based on $\mathcal{T}_m$, the integrity of one or more components or connections is compromised.

We consider only systems whose designs can be expressed as *synchronous* (finite- or infinite-state) transition systems. For such systems, CRV focuses on *integrity* properties, functional properties that can be violated by compromising the integrity of their components or their interconnections. Technically, CRV currently analyzes only system properties that can be expressed as *temporal safety properties* of the form $\square \bigwedge_i a_i \rightarrow \square \bigwedge_j g_j$ where a_i and g_j are quantifier-free, first-order past-time LTL formulas and $\square$ is the *always/globally* operator.

A large set of desired system-level integrity properties can be modeled as temporal safety properties of the form supported by CRV. Examples of such properties for the model in our case study (see Section VI) would be the requirement that the delivery drone never deliver a package to an off-limits location or that it deliver a package to a given drop location only if the location is clear. Many other examples exist in practice (see, e.g., [19], [20], [15]).

Currently outside the scope of CRV are *confidentiality*, *authentication*, and *availability* properties. Traditional analyses of confidentiality properties such as non-interference only consider attacks on system-level inputs, as opposed to attacks on individual components of the system under analysis. Supporting that traditional analysis in CRV would not be difficult. Both the approach and the tool could be easily extended to compose the input system design with itself (i.e., have two copies of the system running in parallel) while asserting the top-level safety property that the public outputs are equal whenever the public inputs are equal. In contrast, in our case study, where we also look at attacks on system components, considering non-interference would require a new class of properties whose violation may include *both* confidentiality and integrity guarantee violations at the component level.

Authentication properties are relevant only to protocols that use cryptographic constructs, as they are easily violated without such constructs. For those cases, a cryptographic protocol verifier could be incorporated in principle into CRV's workflow to enable reasoning about cryptographic constructs. Investigating this integration is left to future work.

Although CRV could potentially reason about availability properties (e.g., under the right conditions, the delivery drone will make a successful delivery), adding this capability would be more challenging in general. The reason is that such properties translate to *liveness* properties, and current model checking technology is not advanced enough to prove (unbounded) liveness for most realistic models of infinite-state systems. However, we intend to provide support for some restricted classes of availability properties in the future.

B. CRV *Architecture and Workflow*

The high-level architecture of CRV and its interaction with the system designer is shown in Figure 1. The designer starts by developing, in a suitable modeling language, the design of the system to be analyzed. Currently, CRV is available [32] through a plug-in of the OSATE IDE for the Architecture Analysis and Design Language (AADL) [21] extended with AGREE [11] contract language. The design describes the system's architecture in terms of system components, component interfaces and interconnections, and a list of components and connections considered vulnerable to attack. In addition, the design also contains behavioral information for some components, expressed in the form of *assume-guarantee* contracts, capturing how input and internal state values are converted to outputs and state updates. Finally, the architect adds system-level properties that the model should satisfy, expressed as guarantees of the top-level component in the design.

When invoked, CRV's *front-end* translates the design and the system level properties into an intermediate representation (IR). CRV can support other modeling languages with the addition of the corresponding IR translators. The IR and one or more user-selected threat models are then fed into the *threat instrumentor module* which modifies the model's IR to include adversarial influences according to the list of vulnerable components and connections. Finally, the threat-instrumented model is fed to the Kind 2 model checker [9],

[1] A VM image containing tool, models, and related instructions is available to the reviewers at: http://clc.cs.uiowa.edu/fmcad23/.

[25] in order to prove or disprove that the model satisfies the desired properties.

For each satisfied property, CRV's *merit assignment module* identifies system components and connections in the design which are critical for satisfying the property. Dually, for each violated property, the *blame assignment module* identifies the vulnerable components/connections that contributed to the violation. An *attack trace*, describing the attacker's behavior and the system's response, is also presented as evidence.

C. Challenges

CRV addresses the following three analysis challenges.

Scalability. The complexity of model checking problems CRV solves ranges from NP-hard to undecidable. CRV addresses this scalability challenge by leveraging Kind 2's reasoning support for complex, hierarchical systems in the form of *compositional* verification where verification results for sub-systems/components are used to discharge verification conditions of higher-level components.

Behavioral modeling. Capturing components' behavior in a design model at the right abstraction level for a successful analysis is a challenging task. More abstract behaviors simplify the automated analysis but can lead to an increased number of *false positives*: execution traces that falsify the property but are not actual executions of the modeled system. In contrast, more detailed behavior decreases or eliminates false positives but can increase the analysis complexity to the point of overwhelming the model checker. We prescribe capturing abstract behavioral information in sufficient detail in the form of *assume-guarantee contracts* for selected components. Such contracts state that as long as a component's environment satisfies the contract's assumptions, the component's behavior will satisfy the contract's guarantees.

Threat instrumentation. The final challenge is how to incorporate the adversarial influence in the design. One possible approach is to explicitly consider concrete attacks (such as buffer overflow, malware attacks, and so on). Unfortunately, this approach has serious shortcomings: (i) listing all possible integrity attacks can be cumbersome and time-consuming; (ii) the attacker model becomes outdated with the discovery of newer attacks; (iii) the design may not contain implementation-level details for the sake of scalability, making it difficult to describe specific attacks (e.g., SQL injection); (iv) no guarantees can be provided against *zero-day* attacks. To address these challenges we consider attack *effects* instead of concrete attacks, as explained in the next section.

III. Design Model Instrumentation

In this section, we explain the notion of attack effects and then discuss the automatic instrumentation of the model.

The Problem. The threat instrumentation process in CRV solves the following main problem: *Given a formal description of a system component in terms of its input/output interface and behavior, how do we capture all possible integrity attacks that can impact the component's behavior?* The problem can be further decomposed into two technical questions: (Q_1) how to consider all possible integrity attacks efficiently; (Q_2) how to incorporate the different attacks, some of which can be implementation-specific, into a design containing only abstract information.

A. Attack Effects

Our main insight for addressing question Q_1 above is to switch attention from attacks to their *effects*. More precisely, we argue that, in the context of resiliency analysis, it is sufficient to consider the *effects of integrity attacks on a system component's behavior* instead of the concrete attacks themselves. Capturing the effects frees us from having to worry about the details of how each attack is achieved in concrete. Effects of integrity attacks can be viewed as *attack abstractions* which group together integrity attacks according to their consequences on a system. In addition to simplifying reasoning, this sort of abstraction allows us to reason at once about *all* integrity attacks having a certain set of effects.

In general, an integrity attack can have the following three effect types: (E_1), or *standard attacker*, lets the adversary modify the behavior of the attacked component at will—as done, for instance, in buffer overflow attacks; (E_2), or *replay attacker*, lets the adversary replay previous values of data sent across vulnerable network connections; and (E_3) which can render the component unresponsive—for instance, by making it crash. Note that (E_1) is strictly stronger than (E_2)— the user chooses between (E_1) and (E_2) based on the system and adversary under consideration. Both cases include (E_3). In our context, these effect classes are sufficient to account for any kind of integrity attack in a system design.

Capturing Attack Effects In view of our classification of attack effects, question (Q_2) becomes how to incorporate the three types of effects (E_1–E_3) into the design *automatically*. When instrumenting a component or connection with effects of type (E_1), CRV does not constrain in any way the behavior an adversary would choose. This allows it to reason about *all relevant* attack strategies that violate the desired functional properties. When instrumenting a connection with effects of type (E_2), CRV constrains the adversary's behavior to only inject values that were previously sent along the connection.

Technically, we model adversarial actions by adding *nondeterminism* in selected places in the model, accounting for unconstrained adversary behavior, and letting the model checker find a concrete behavior (*i.e.*, an execution trace) that results in a property violation. In essence, we let the model checker play the adversary by allowing it to replace the output of a compromised component by any value (of the correct type) of its choosing or, in the case of a replay attacker, by any previous output value.

In terms of the Dolev-Yao model [13], a formal model common in cryptographic protocol verification, CRV effectively places a *restricted* Dolev-Yao-style adversary after each output of a vulnerable component, allowing the adversary to: arbitrarily change an output, mimicking (E_1) attacks; replay a previous value, mimicking (E_2) attacks; or drop an output, mimicking (E_3) attacks. Note that the standard Dolev-Yao

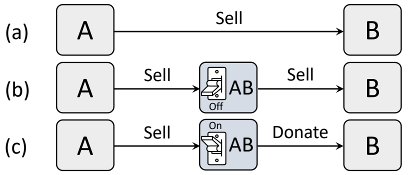

Fig. 2: (a) Non-instrumented channel between A and B; (b) instrumented but not enabled channel between A and B; and (c) enabled instrumented channel between A and B.

model considers a *network adversary* to be placed only on *public channels* where it has the following capabilities: (a) it can sniff, (b) drop, and (c) modify any message passing through the channels; (d) it can send messages impersonating the protocol participants; moreover, (e) it can exercises capabilities (a)–(d) while conforming to cryptographic assumptions. Our restriction of the model does not include capabilities (a) and (e) as CRV does not currently consider confidentiality properties nor reasons about cryptographic constructs.[2] Our instrumentation further deviates from the Dolev-Yao model by placing an adversary in a *non-public* channel (i.e., encrypted and integrity protected) when the sender in the channel is marked by the designer as vulnerable to integrity attacks.

As an example, consider the design fragment shown in Figure 2(a) where component A feeds its single output, of enumeration type $\{\mathsf{Sell}, \mathsf{Donate}\}$, only to component B. If A is vulnerable to integrity attacks (e.g., buffer overflow) *impacting its integrity*, then we instrument the model by placing a new component AB between A and B, effectively simulating a vulnerable public communication channel as in the Dolev-Yao model. The new component has two inputs, namely the output of A and a Boolean input corresponding to an enabling switch (analogous to an activation variable in fault analysis), and a single output sent as input to B. When the switch is off (see Figure 2(b)), the AB component behaves as a benign lossless channel, faithfully forwarding the output of A to the output of B. When the switch is on, the AB component can behave maliciously by replacing the output of A with a different value chosen non-deterministically (see Figure 2(c)). We use the switch inputs because they are essential for generating diagnostic information (*i.e.*, blame assignment), where each switch is treated symbolically as a model parameter.

B. Utility of CRV*'s What-if Analyses and Threat Models*

CRV comes with a set of user-selectable built-in threat models, which allow it to determine automatically vulnerable components and connections. To use these, the designer annotates model components and connections with a few security-related meta-level attributes, for example, the pedigree of a component expressed by an enumerated type like $\{\mathsf{COTS}, \mathsf{Sourced}, \mathsf{inHouse}\}$. (See [14] for a detailed list of such meta-level attributes.) Then, CRV identifies components and connections that should be considered vulnerable to attack in the selected threat model(s). The details of the built-in threat models in CRV are presented in Section III-C.

Based on our presentation so far, a system designer can pose *what-if* queries of the sort: *What happens if a set X of vulnerable components and connections is subject to integrity attacks? Does the system design still ensure the satisfaction of the desired properties?* To illustrate the utility of the underlying *what-if* analyses of CRV, we describe how a system architect (i) could model *supply chain attacks* and (ii) design *zero trust networks*.

To model a *supply chain attack* [29], the system designer annotates each model component corresponding to an outsourced subsystem as vulnerable to attack. Given this information, CRV determines if violations of the outsourced component's contract (in this case, due to supply chain vulnerabilities causing the component to misbehave) lead to violations of system-level integrity properties. If so, the designer should consider mitigating actions such as producing the subsystem component in-house or pursuing additional supply chain protections. If not, the model is resilient to supply chain attacks.

Zero trust networks are networks where every component performs input validation, rather than assuming that internal network connections are safe from attack [28]. As input validation can be computationally expensive, a system designer might want to verify if validation can be safely skipped for some input channels. To model this situation, the designer marks the corresponding model connections as vulnerable, and CRV will report if attacks on such connections affect system-level properties. If no system-level properties are violated under the chosen threat model(s), the system is cyber-resilient enough for the designer to consider forgoing validations steps on inputs coming through the marked connections.

C. Built-in Threat Models

CRV can automatically identify the vulnerable model components and connections according to some built-in threat models. A *threat (effect) model*, in this context, conservatively describes the criteria under which certain components and channels can be considered vulnerable to attacks (such as, logic bomb, remote code injection, and malware) that impact the control-flow integrity of the components/channels matching the criteria.

More operationally, we express threat models as queries on the design model data that specify the criteria for classifying components or channels as vulnerable to certain integrity attacks. Concretely, a specific threat model can be expressed as a query whose result is a set of components/connections that satisfy specific constraints on the meta-level attributes.

As an example, a component is automatically classified as susceptible to logic bomb or software Trojan attacks if (i) the component's type is software or software hybrid; (ii) its pedigree is either `COTS`, or `Sourced` without supply chain protection or tamper protection; and (iii) the component has not gone through static analysis or adversarial testing for logic bombs. A list of threat model descriptions currently used by CRV is given in [14]. Note that it is easy to expand such a list

[2] Capability (e) can be simulated by incorporating a cryptographic protocol verifier into CRV's workflow.

since new meta-level attributes and threat model descriptions can be added modularly.

IV. Model Checking and Diagnostics

We now discuss how CRV takes advantage of automated reasoning to perform resiliency analysis and generate meaningful traceability and diagnostic information from it.

Model Checking for Resiliency Analysis. We reduce the problem of checking the resiliency of a system design to integrity attacks to a model checking problem for the threat-instrumented version of the design. In CRV's workflow, the threat instrumented model and the desired functional properties are fed to the Kind 2 model checker [9]. Using induction-based techniques, Kind 2 tries to prove that each property is satisfied for any possible execution, including those containing the attacks contemplated by the instrumentation. In parallel with that, Kind 2 uses bounded model checking techniques to exhaustively search for execution traces in the instrumented model that violate one or more of the given properties. For each property, Kind 2 can output three possible verification results: SAFE, meaning that the instrumented design satisfies the property; UNSAFE, meaning that the design allows executions violating the property; and UNKNOWN, returned for instance when the model checker times out. For each definite answer (SAFE or UNSAFE), Kind 2 provides additional diagnostic information, as discussed below. The UNKNOWN case is due to the undecidability of the model checking problem for infinite-state systems in general and to the high computational complexity of the problem in decidable subcases. We emphasize that Kind 2 employs *sound* proving techniques which are, however, necessarily *incomplete* in the case of infinite-state systems.

End-to-End Security Properties. A designer may wonder if they could simply model each component in isolation, rather that dealing with a complex, hierarchical model spanning the entire system. Thanks to CRV the latter strategy is preferable as it enables the designer to reason formally about end-to-end properties that reference output from *multiple components*. To start, end-to-end reasoning enables the user to prove that system-level properties hold in the benign case, which is an important initial sanity check. More important, CRV's analysis may show that property violations for one or more individual sub-components do not actually lead to system-level attacks — which is the case when CRV proves that system-level properties are still preserved. In contrast, without a tool like CRV, the designer has to *manually* reason about whether attacks on individual components can compose to a violation of a system-level property.

A. Attack Traces

For each property that it proves UNSAFE, the model checker returns as evidence to CRV an input/output counterexample trace, in effect an attack trace. The trace contains detailed information of the attacker's actions (*i.e.*, the non-deterministic choices made in the instrumented channels) as well as the reactions of the other components to those actions.

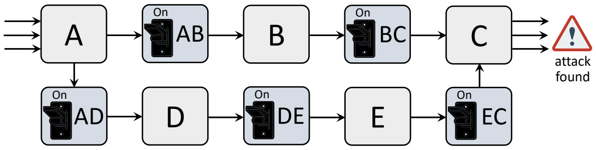

Fig. 3: An example instrumented design with all components being vulnerable.

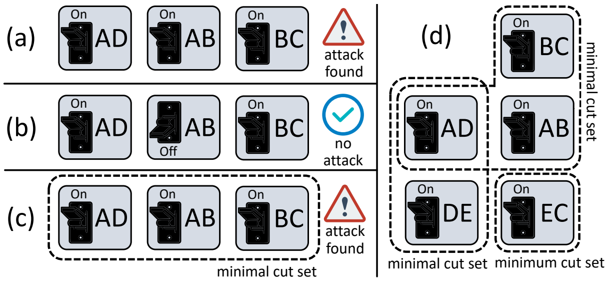

Fig. 4: An example of minimal and minimum cut-set.

B. Blame Assignment

For each property determined to be UNSAFE, in addition to the attack trace, CRV can also generate information regarding misbehaving components that may have contributed to the violation. We call this functionality *blame assignment*. CRV supports a *locally optimized* [24] and a *globally optimized* [23] form of blame assignment, both achieved by posing a series of queries to the backend model checker.

To understand blame assignment, consider as an example the threat-instrumented system design sketched in Figure 3 where all components are vulnerable according to the threat model specified by the user. This is reflected by the presence of an adversarial component (*e.g.*, AB) between each pair of connected components (*e.g.*, A and B). Each adversarial component is switched on, that is, it is enabled to perturb the communication between the components it links. Once the model checker finds an attack trace that demonstrates the violation of a property, the blame assignment module of CRV will try to minimize the number of enabling switches that must be turned on to cause a violation of the property. Technically, this is analogous to finding a *minimal cut set* [1], [23] for these switches. Suppose it is enough to turn on just components AD, AB, and BC in Figure 3 for the model checker to come back with an UNSAFE verdict for the property (see Figure 4(a)). Then these three switches form a minimal cut set only if turning off any of them (say, switch AB) changes the verification verdict to SAFE (see Figures 4(b)). Since there may be many different minimal cut sets for the complete set of switches, CRV tries to find the one with the smallest cardinality (*e.g.*, {EC} in Figure 4(d)).

C. Merit Assignment

If, after its resiliency analysis, the model checker returns a SAFE verdict for a given desired property, CRV also provides a list of components whose behavior *might be* critical for the satisfaction of the property. The conditional is necessary

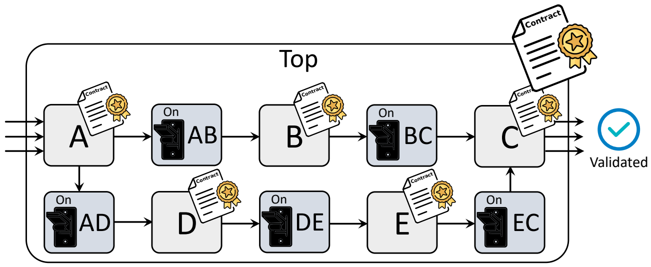

Fig. 5: The top component's contract is respected given all internal components' contract are validated.

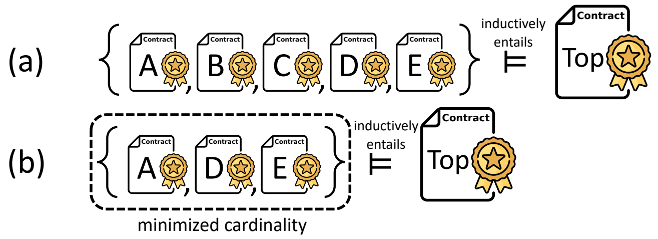

Fig. 6: Merit assignment problem as finding a minimal subset of the set in (a) that still inductively entails a given guarantee in the system's contract.

because, for efficiency reasons, the user has the option of requesting an over-approximation of the critical list. This functionality called *merit assignment* is useful for two reasons: (i) it provides better traceability in the model by confirming whether the defenses put in place by the designer in response to a (previous) violation of the property indeed play a role in maintaining the property; (ii) it provides additional information for system developers, who then know that behavioral changes to the components outside the merit assignment set will not affect the property and that, instead, extra precautions should be taken when implementing components in that set.

Merit assignment in CRV relies on the computation of a *minimal inductive validity core* (MIVC) [17], another functionality provided by Kind 2. As in other symbolic model checkers, Kind 2 represents an input model internally as a transition system consisting of an initial state predicate I characterizing the system's initial state(s) and a two-state transition relation T describing the system's behavior. The relation T can be expressed as conjunction or, equivalently, a set of constraints over states and their successors. A MIVC for a particular property P satisfied by a transition system (I, T) is a minimal subset T' of T such that (I, T') also satisfies P. The name MIVC comes from the fact that the model checker proves that a transition system satisfies P, *i.e.* that the property is *valid* in every execution of the system, by using *inductive* arguments. In that case, one can say that the system *inductively entails* the property.

To illustrate the concept, let us consider the example in Figure 5. Suppose the model checker is able to prove that the composition of the top component's behavior and that of each of its subcomponents, expressed by its contract, inductively entails a desired system-level property, expressed as a guarantee in the top-level contract (see Figure 6(a)). Merit assignment provides a minimal subset of subcomponents whose contracts suffice for the proof. Concretely, if the contracts of the subcomponents A, D, and E are both sufficient and necessary to construct a proof of the guarantee (see Figure 6(b)), then A, D, and E, and only those, will be included in the merit assignment. Furthermore, for each included subcomponent, CRV will single out the individual guarantees in the subcomponent's contract that are enough to prove the desired top-level property.

In our case, due to inherent runtime complexity reasons, we resort to identifying *approximate MIVC*, *i.e.* (not-necessarily minimal) supersets of a MIVC, another functionality provided by Kind 2. Experimental results by Larraz *et al.* [23] show that approximate MIVC typically approximate true MIVC very closely while requiring significantly less time to compute.

V. Implementation of CRV

Although designed to be incorporated in various modeling environments, CRV is currently available as a functionality of VERDICT, a larger cyber-resilience analysis tool developed with partners at GE Research [27]. In this section, we provide some details specific to CRV's implementation.

Modeling Language. We instantiated CRV for the AADL modeling framework [16]. An AADL model can capture the architecture of a synchronous reactive system in terms of components and their interconnections. *Components* of an AADL model are *systems*, which group together other components (or subsystems), and *data*, which define the data types used by the various systems. To model interactions among (sub)systems one defines an interface for each of them consisting of *data ports*, *event ports*, and *connections*. AADL can be extended by users in two ways. The first is the addition of user-defined attributes, called *properties* in AADL. This allows us to capture whether components or their connections should be considered vulnerable to attack. AADL also has an extension mechanism based on *language annexes* with which one can embed a domain-specific language (DSL) in AADL and use it to enrich the design description. One such annex contains the AGREE DSL [11] which allows one to express behavioral information for synchronous components formally and declaratively. Component behavior is specified in AGREE either as an assume-guarantee contract or as an *implementation* consisting of equational constraints on ports and internal state variables.

Front-end Translator. We developed a translator for CRV, written in Java, that takes as input an AADL design model enriched with security-related AADL properties and component-level behavioral specs in AGREE, and translates it to an intermediate textual representation (IR). Our IR is general enough to accommodate a wide variety of modeling languages supporting the synchronous model of computation (*e.g.*, Simulink+Stateflow). The IR is close in structure to the synchronous dataflow language Lustre [18].

Threat Instrumentor. The threat instrumentor module of CRV is written in Java. It takes the IR version of the design model and the selected list of vulnerable components and connections, and returns a threat-instrumented design, also written in the IR language.

Standard Attacker. When CRV generates the instrumented model, a component is automatically generated to specify the adversary's behavior. A standard attacker, such as in Figure 2(c), is modeled as an intermediate component which takes messages from component A as input and produces adversarially-instrumented messages to pass to component B.

Bounded Replay Attacker. A replay attacker component is obtained by adding a contract to a standard attacker that restricts its output messages to be equal to previous (legitimate) messages from the last n time steps. This models adversaries that can only replay past outputs and have a bounded memory. The number n is configurable by the user in CRV's front end.

Unbounded Replay attacker. We also include support for a replay attacker with unbounded memory, i.e., the ability to replay messages that were sent arbitrarily far in the past. This is achieved by representing the sequence of past outputs in the model as an uninterpreted function from time steps to output values that is progressively constrained at each step with the current output value, and by allowing the model checker to query the function at any step up to the current one.

Model Checker. As already discussed, CRV uses the Kind 2 model checker in the backend for its analysis. An internal translator in CRV, not shown in the architecture in Figure 1, translates the instrumented IR to a system model written in Kind 2's input language, an extension of Lustre with support for assume-guarantee contracts [8]. CRV then asks Kind 2 to prove the correctness of the top-level component in the model, standing for the entire system, with respect to its contract. Kind 2 returns its results to CRV incrementally as it proves or disproves each top-level (*i.e.* system-level) guarantee. In turn, CRV converts those results in terms of diagnostic information on the input AADL model and provides it to the user.

Diagnostic Information. The basic level of diagnostic information tells the user if each system-level guarantee is satisfied, violated, or undetermined (because of a timeout). It also provides a counterexample trace for each violated guarantee. The next level includes blame assignment for the violated guarantees and merit assignment for the satisfied ones. We implemented the bulk of the blame and merit assignment functionality directly as an extension of Kind 2, which is written in OCaml. The two features collectively span over 2K lines of OCaml code. For locally optimized blame assignment, we rely on the MaxSMT functionality provided by the Z3 SMT solver [12].

OSATE Plugin. We provide CRV's functionalities through a plugin developed with our industrial collaborators [32] for OSATE [31], a development and analysis environment for AADL models.

VI. A Case Study on Unmanned Delivery Drone

We now discuss a case study on analyzing a realistic model of a hypothetical unmanned delivery drone (UDD).

Goal. The case study had the following objectives: (i) provide evidence that CRV can analyze complex designs; (ii) concretely illustrate the responsibility of the human designer in the CRV workflow (*i.e.*, the manual steps); (iii) show the interaction of the system designer with CRV; (iv) assess the value of CRV's blame and merit assignment features for debugging the design; (v) evaluate CRV's runtime performance.

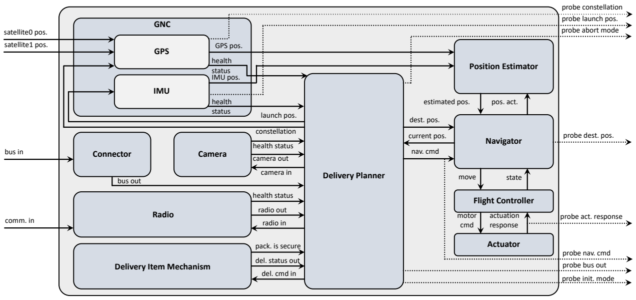

Fig. 7: System architecture of unmanned delivery drone (UDD)

High-level UDD system description. The drone is a part of a last-mile delivery unit consisting of a van with packages to be delivered to suburban locations, and one or more delivery drones, also stored in the van. Once the van arrives at a location close to multiple delivery sites, each delivery drone is initialized with its current position and delivery location, and is loaded with the package for that delivery location. Once the drone takes off with the package, it uses inputs from a GPS receiver and an Inertial Measurement Unit (IMU) to navigate to the delivery location. When it reaches the delivery location, the drone uses an on-board camera to capture an image of the delivery site to confirm that the landing location is clear and so it is safe to drop the package. For high-value packages, the delivery drone uses radio communication to get confirmation from the operator in the van. If there are no obstacles in the delivery location and a confirmation (if needed) is received from the operator, the drone's controller activates a delivery mechanism to drop off the package. The drone then returns to the van for another delivery or storage.

Scenarios. For our case study, we consider two scenarios and 7 functional properties to demonstrate CRV's effectiveness. As a result, we identified one design weakness/attack for each property. In Sections VI-A through VI-C we focus on one scenario consisting of a single property. The remaining scenarios and properties are presented in Section VI-D.

A. *System Architecture of* UDD

The system designer first develops the architecture of the UDD, which can be visualized graphically with a diagram like the one in Figure 7. In AADL, this is done by defining the top-level component and each of its subcomponents as individual AADL systems, and then specifying their interface and connections. As an example, here is a specification of the interface of the `DeliveryItemMechanism` system in our model:

```
system DeliveryItemMechanism
 features
  delivery_cmd_in: in data port PackageDeliveryCmd;
  delivery_status_out: out data port DeliveryStatus;
  package_is_secure: out data port Boolean;
end DeliveryItemMechanism;
```

For each component, the designer identifies a set of input and output ports used by the component to communicate with its environment, and specifies the type of data exchanged in each port. In addition to basic types, such as `Boolean` and `Integer`, AADL allows the use of user-defined types for ports.

After that, the architect can describe the internal structure of each composite system, by adding a *system implementation* (not shown here) that lists the system's subcomponents and specifies how they are connected together.

B. Design Model of UDD

Next, the designer specifies the behavior of each *leaf* component of the architecture, *i.e.*, a component with no subcomponents. This is achieved by associating to it an assume-guarantee contract written in AGREE. Intuitively, assumptions describe the expectations the component has on its inputs and on the global values it has access to, while guarantees describe restrictions on the output values it produces. Behaviorally, each component is a reactive system, instantaneously producing output based on its current input and internal state. Assume-guarantee contracts in AGREE are essentially statements in a linear temporal logic rich enough to precisely describe a component's reactive behavior from the point of view of an observer that, at all times, has access to all the input and output values generated until then. Atomic formulas in this logic are first-order predicates that relate the values of the various ports and intermediate variables. Contracts provide a mechanism for capturing the information needed to specify and reason about component-level safety properties at the desired level of abstraction. Now, let us assume that for the `DeliveryItemMechanism` component the following is known:

1) Initially the delivery has *not started*.
2) If a delivery command is issued, the delivery status must become different from *not started*.
3) If no command is issued or an *abort* command is received, then the delivery status resets to *not started*.

This is a minimal amount of information about the expected behavior of the `DeliveryItemMechanism` component. We explain how to formalize it in AGREE below, using abstract syntax for conciseness.

To formalize (1), we add the following guarantee stating that the delivery status s (abbreviating `delivery_status_out`) is equal to not_started initially:

$$G_1: \quad \mathsf{s} = (\mathsf{not\_started} \rightarrow \mathsf{true})$$

The infix initialization operator → is an AGREE operator. An expression of the form $e_1 \rightarrow e_2$ evaluates to the value of expression e_1 initially and to the value of e_2 at all later steps of the system's execution. To formalize aspect (2), we add the following guarantee where c abbreviates `delivery_cmd_in`:

$$G_2: \quad \mathsf{true} \rightarrow (\mathsf{c} = \mathsf{release} \Rightarrow \mathsf{s} \neq \mathsf{not\_started})$$

Similarly, we capture aspect (3) with this guarantee:

$$G_3: \quad \mathsf{true} \rightarrow \\ (\mathsf{c} = \mathsf{no\_op} \vee \mathsf{c} = \mathsf{abort} \Rightarrow \mathsf{s} = \mathsf{not\_started})$$

So far, we have only added constraints about the output port `delivery_status_out`. Any system execution that satisfies those constraints will be considered valid during the analysis performed by CRV.

Desired Functional Requirements. The next step is to review the list of functional (or *safety*) requirements for the system that may affect its integrity, and formalize them as cyber-resiliency properties in AGREE. For instance, suppose we have the following cyber-requirement for `DeliveryDroneSystem` which forbids package delivery to certain locations (while allowing the UDD to still fly-over them):

P7. *The drone will never initiate a packet release in an off-limits location.*

To formalize P7, we have to first identify the components and ports of the system that are relevant to this property. In our example, `DeliveryPlanner` is the component that issues the command to release a package by setting the output port `delivery_cmd` to release, while `DeliveryItemMechanism` is the component that receives the command and proceeds with the delivery. Moreover, to know where the drone should release the package, the `DeliveryPlanner` reads the delivery location from the input port bus_in through the `Connector` component when the drone is in the van, and then it passes this value to the `Navigation` component. In addition, we also need to know when a location is off-limits. For that, we can define a new predicate InRA over locations that evaluates to true if and only if its input location is within a restricted area.

Then we can express the cyber-property with the following top-level guarantee where dl is the delivery location:

$$P_7: \quad \mathsf{InRA}(\mathsf{dl}) \Rightarrow \mathsf{s} = \mathsf{not\_started}$$

Vulnerable Components and Connections. To make the design amenable to CRV's analysis, designers also need to annotate components and links that are vulnerable to attack. For instance, suppose that `DeliveryPlanner` is considered vulnerable to attack (*e.g.*, it is an outsourced software component that has no supply chain security and no tamper protection, and has not been statically analyzed).

C. CRV *Analysis*

Analysis in the Benign Case. The designer must check that the system design model satisfies its guarantees in the *benign scenario* where no threat models are enabled. For our example, CRV finds an execution that violates P7 because the delivery location information provided as input during initialization is actually an off-limits location. In this case, there are two possibilities: (S_1) the designer decides to prohibit that initially provided delivery locations be off-limits; (S_2) the designer decides to treat initialization with an off-limit location as a realistic possibility, perhaps as a consequence of malicious code in the (external) software (in the van) that provides initialization values for the UDD system. Assume (S_2) cannot happen in the benign scenario. Then the designer adds the following assumption to the contract of `DeliveryDroneSystem` to capture (S_1) where tl is the target location provided through the system's input bus:

$$A_1: \quad \neg\mathsf{InRA}(\mathsf{tl})$$

In this case, with no threats enabled, CRV is able to prove property P7 valid.

Analysis under Threat Effects. After verifying with CRV that P7 holds in the benign case, the system designer can run an adversarial analysis. In this case, CRV will find a violation of property P7 that involves an attack to the UDD's delivery planner. From the blame assignment diagnostics, we observe that the possible violation may result from an attack on the `DeliveryPlanner` component that causes it to violate its contract. The blame assignment analysis additionally identifies a minimal set of ports, `dest_location` and `delivery_cmd`, that is enough to compromise to carry out the attack successfully. One can also examine the counterexample trace leading to the violation of the property and observe that it occurs when the vulnerable `DeliveryPlanner` maliciously instructs the `DeliveryItemMechanism` component to initiate the release of the package while the drone is flying over an off-limits location.

Analysis after Mitigation. After CRV presents a new attack, the designer can see how to address its root cause by considering the vulnerable components and ports relevant to the attack. Then, they can run another *what-if* analysis by considering a situation where additional security measures have been introduced to make some of the vulnerable components more cyber-resilient. For example, suppose the designer sees the `DeliveryPlanner` component as a candidate for enhanced cyber-resiliency measures. When the component is labeled as invulnerable to attack, a new analysis of the system confirms that this fix is sufficient to rule out any attacks compromising property P7. Moreover, the merit assignment post-analysis reassures the user that the change indeed plays a role in the satisfaction of P7 in an adversarial environment.

Performance. The AADL+AGREE model for the UDD system in this case study consists of around 1800 lines of specs, and includes 7 functional properties as top-level guarantees. On average, each call to the tool, which triggers the threat instrumentation, the verification of the properties, and the merit/blame assignment analysis, takes 30s on a 1.10GHz Intel(R) Core i7-10710U CPU machine with 16GB of RAM.

D. Details on Case Study Experiments

We used CRV to analyze the cyber-resiliency of the UDD system with respect to two sets of system-level safety properties. The first set consists in the following five properties:

```
guarantee "P1: When the drone is switched on,
  the GPS component uses the constellation
  received most recently":
  isOn =>
    most_recent_constellation = probe_constellation;

guarantee "P2: Launch location for IMU is
  initialized properly":
  isOn =>
    most_recent_launch_loc = probe_launch_loc;

guarantee "P3: Delivery location for navigation
  is initialized properly":
  isOn =>
    most_recent_delivery_loc = probe_delivery_loc;

guarantee "P4: A command to release a valuable
  package is issued only if drone has received
  confirmation from base":
  release_cmd and valuable_package =>
    target_confirmed;

guarantee "P5: The drone will always request
  confirmation from base before starting delivery
  of a valuable package":
  delivery_started and valuable_package =>
    confirmation_requested;
```

First, we checked the system model satisfies the five properties without considering the effects of any threat model. CRV was able to prove that in 9s. Then, we analyzed the cyber-resiliency of the system against all threats in the CRV library. As a result, CRV was able to find in less than 5s four network injection attacks on `bus1` leading to the violation of properties P1, P2, P3, and P4, and one logic bomb attack on the `DeliveryPlanner` causing the violation of property P5. Note this is just one possible blame assignment result, the model admits other minimal results. After reviewing the results, we considered a scenario where the four network injection attacks were not possible because the `bus1` connection was a *trusted* connection, and the logic bomb attack was not feasible anymore after the decision to develop the `DeliveryPlanner` internally instead of outsourcing it. To reflect the new scenario we changed the meta-level attribute `connectionType` of the `bus1` connection from `Untrusted` to `Trusted`, and the meta-level attribute `pedigree` of the `DeliveryPlanner` component instance from `Sourced` to `InternallyDeveloped`. After the change, we could check that no more new attacks were possible by analyzing again the modified model against all threats in the CRV library. This time CRV proved all five properties valid in 20s.

The second set consists in the following two properties:

```
guarantee "P6: The drone issues a command to
  release a package only if the delivery location
  is the most recent delivery location provided":
  release_cmd =>
    probe_delivery_loc = most_recent_delivery_loc;

guarantee "P7: The drone never initiates a package
  release to an off-limits location":
  delivery_status <> NOT_STARTED =>
    not InRestrictedArea(probe_delivery_loc);
```

In this new scenario, we considered the values of the meta-level attributes for the `bus1` connection and the `DeliveryPlanner` component instance to be the original ones. Again, we started checking that properties P6 and P7 were satisfied by the system, which CRV confirmed after 8s. Then, we focused on analyzing the cyber-resiliency of the system with respect to property P7. Similarly to the first scenario, CRV was able to detect in 3s a logic bomb attack on the `DeliveryPlanner` that leads to the violation of P7. We chose to neutralize the attack by implementing a runtime monitor, as explained in Section VI, instead of enforcing the internal development of the component. After integrating the behavioral defense in the model, CRV could prove the satisfaction of property P7 in 3s. The next step was to analyze the cyber-resiliency of the system with respect to property P6. In 3s, CRV confirmed that a logic bomb attack on the `DeliveryPlanner` would still be able to falsify property P6. In this case, we decided to apply a hybrid solution. First, we forced the `DeliveryPlanner` component to be internally developed to prevent logic bomb attacks. Then, we changed the model design to incorporate a MAC protection, to make the system cyber-resilient to network injection attacks on the `bus1` connection. After those changes, CRV could prove the satisfaction of property P6 and P7 in 168s.

VII. Related Work

There are at least four relevant lines of work that analyze system designs for security threats. They are based respectively on: general model checkers, cryptographic protocol verifiers, fault analysis tools, and specialized analyzers. We highlight the main differences between CRV and each of these classes of tools.

A. Model Checkers

Model checkers used for cyber-security generally perform analysis at one of two granularities: system-level and component-level. The former considers only system-level inputs to be adversarially-controlled, overlooking cases where the integrity of individual components is violated. The latter considers inputs of an individual component to be adversarially-controlled, letting one mimic the integrity violation of that component. To lift the analysis from individual components to the whole system, manual efforts are needed to recompose component-level guarantees into global/system-level properties. Correspondingly, component-level counterexamples have to be lifted to the system-level in order to construct a system-wide attack. In contrast, the results obtained from the analysis of the threat-instrumented model in CRV can be automatically interpreted at the system level.

Instrumented versions of a system model are used in traditional fault analysis to study the system's behavior in the presence of faults. The Safety Annex for AADL [30] allows one to specify the behavior of systems and components in the presence of faults. The tool supports the computation of *all* minimal cut sets, but not the direct computation of an individual solution. Similarly, the xSAP [4] platform offers library-based specification of faults, automatic model-extension with fault specifications, and the generation of minimal cut sets. However, its primary emphasis also lies in fault analysis rather than security. Finally, all these techniques focus on the globally optimized setting. To our knowledge, the use of the locally optimized setting [24] is novel.

B. Cryptographic Protocol Verifiers

Cryptographic protocol verifiers (CPVs) such as Tamarin [26] and ProVerif [5], [22], [2] can reason about cryptographic constructs and support confidentiality properties (e.g., observational equivalence), neither of which are currently supported by CRV. In contrast, CRV has the following advantages: (i) it can support rich system descriptions with linear (integer and real) arithmetic constraints and temporal constraints, which are not supported by current CPVs; (ii) it can provide diagnostic information in the form of blame and merit assignment at the end of the analysis, which is unavailable in CPVs; (iii) it can analyze rich stateful systems more scalably than current CPVs; (iv) it supports automated analysis of rich safety properties beyond what is supported by tools such as ProVerif [5] or their extensions [22], [2]; (v) thanks to compositional verification, it can support the analysis of large systems that are not amenable to analysis by CPVs. In a sense, CRV and CPVs are complementary. One way to support confidentiality properties and cryptographic constructs in CRV's workflow would be by integrating a CPV in it.

C. Fault analysis tools

Fault analysis tools, especially the ones that consider Byzantine faults, are the closest in spirit to the CRV work. However, they *generally* are neither amenable to seamless integration with CPV (due to the lack of support for replay attackers), which is needed to analyze rich properties of systems containing cryptographic constructs, nor do they support meta-level diagnostic analyses.

D. Specialized analyzers

These analyzers focus on analyzing specific protocols in a particular domain (*e.g.*, cellular networks [19], [20], [3], TCP/IP [6], WiFi [33]), a very limited set of security properties (e.g., non-interference, cache side channel, transient execution vulnerabilities), or particular systems (*e.g.*, IoT [7]). Among these efforts, the closest to our approach are LTEInspector [19] and 5GReasoner [20] where the Dolev-Yao adversary model is used to perturb the public communication between two components when model checking the protocol under analysis. Other tools such as ThreatGet [10] only analyze systems at the architectural level with a pre-defined set of threats. Besides the restriction to specific domains, none of the prior work is capable of statically analyzing reactive system designs with respect to integrity properties — the focus of CRV's analysis.

VIII. Conclusion

We have presented CRV, a general approach and tool to statically check the cyber-resiliency of a design against current and *future* integrity attacks. A case study with an unmanned delivery drone system demonstrates that CRV can analyze effectively the cyber-resiliency of complex designs with respect different integrity properties.

Possible future directions of research include enhancing CRV's capability to support a limited form of *availability properties*, which can be formalized as liveness properties, and *confidentiality properties*, formalizable as non-interference properties. This would enable CRV to support the analysis of functional properties whose violation require a combination of integrity, availability, and confidentiality attacks. Additionally, CRV could be extended to integrate a CPV and hence consider cryptographic constructs.

Acknowledgments

This work was partially supported by DARPA grant #N66001-18-C-4006 and by the US Air Force Research Lab.

References

[1] Parosh Aziz Abdulla, Johann Deneux, Gunnar Stålmarck, Herman Ågren, and Ove Åkerlund. Designing safe, reliable systems using scade. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, First International Symposium, ISoLA 2004, Paphos, Cyprus, October 30 - November 2, 2004, Revised Selected Papers*, volume 4313 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2004.

[2] Myrto Arapinis, Joshua Phillips, Eike Ritter, and Mark D. Ryan. Statverif: Verification of stateful processes. *Journal of Computer Security*, 22(5):743–821, July 2014.

[3] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 1383–1396, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xsap safety analysis platform. In Marsha Chechik and Jean-François Raskin, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 533–539. Springer, 2016.

[5] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 1(1–2):1–135, October 2016.

[6] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, and Paul Yu. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 211–224, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Z. Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 147–158, 2018.

[8] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. CoCoSpec: A mode-aware contract language for reactive systems. In Rocco De Nicola and Eva Kühn, editors, *Proceedings of the 8th International Conference on Software Engineering and Formal Methods, Vienna, Austria*, volume 9763 of *Lecture Notes in Computer Science*, pages 347–366. Springer, 2016.

[9] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 model checker. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517. Springer, 2016.

[10] Sebastian Chlup, Korbinian Christl, Christoph Schmittner, Abdelkader Magdy Shaaban, Stefan Schauer, and Martin Latzenhofer. THREATGET: towards automated attack tree analysis for automotive cybersecurity. *Inf.*, 14(1):14, 2023.

[11] Darren Cofer, Andrew Gacek, Steven Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In Alwyn E. Goodloe and Suzette Person, editors, *NASA Formal Methods*, pages 126–140, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[14] Michael Durling, Heber Herencia-zapana, John Interrante, Baoluo Meng, Abha Moitra, Kit Siu, Vidhya Tekken Valapil, Daniel Prince, Cesare Tinelli, Omar Chowdhury, Daniel Larraz, Moosa Yahyazadeh, and Fareed Arif. DARPA: Cyber Assured Systems Engineering (CASE) — VERDICT Project, 2020. Available at https://github.com/ge-high-assurance/VERDICT/wiki.

[15] Mitziu Echeverria, Zeeshan Ahmed, Bincheng Wang, M. Fareed Arif, Syed Rafiul Hussain, and Omar Chowdhury. PHOENIX: device-centric cellular network protocol monitoring using runtime verification. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[16] P. H. Feiler, B. A. Lewis, and S. Vestal. The sae architecture analysis design language (aadl) a standard for engineering performance critical systems. In *2006 IEEE Conference on Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1206–1211, Oct 2006.

[17] Elaheh Ghassabani, Andrew Gacek, and Michael W Whalen. Efficient generation of inductive validity cores for safety properties. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 314–325, 2016.

[18] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[19] Syed Rafiul Hussain, Omar Chowdhury, Shagufta Mehnaz, and Elisa Bertino. LTEInspector: A Systematic Approach for Adversarial Testing of 4G LTE. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society. tex.ids: hussainLTEInspectorSystematicApproach2018a.

[20] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 669–684, New York, NY, USA, 2019. Association for Computing Machinery.

[21] Software Engineering Institute. AADL – Architecture Analysis and Design Language. http://aadl.info. Accessed: May 20, 2023.

[22] Nadim Kobeissi, Georgio Nicolas, and Mukesh Tiwari. *Verifpal: Cryptographic Protocol Analysis for the Real World*, page 159. Association for Computing Machinery, New York, NY, USA, 2020.

[23] Daniel Larraz, Mickaël Laurent, and Cesare Tinelli. Merit and blame assignment with Kind 2. In Alberto Lluch-Lafuente and Anastasia Mavridou, editors, *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*, volume 12863 of *Lecture Notes in Computer Science*, pages 212–220. Springer, 2021.

[24] Daniel Larraz and Cesare Tinelli. Finding locally smallest cut sets using max-smt. *ACM SIGAda Ada Letters*, 42(2):32–39, Apr 2023.

[25] Daniel Larraz, Arjun Viswanathan, Cesare Tinelli, and Mickaël Laurent. Beyond model checking of idealized Lustre in Kind 2. *ACM SIGAda Ada Letters*, 42(2):40–44, Apr 2023.

[26] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 696–701, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] Baoluo Meng, Daniel Larraz, Kit Siu, Abha Moitra, John Interrante, William Smith, Saswata Paul, Daniel Prince, Heber Herencia-Zapana, M. Fareed Arif, Moosa Yahyazadeh, Vidhya Tekken Valapil, Michael Durling, Cesare Tinelli, and Omar Chowdhury. VERDICT: A language and framework for engineering cyber resilient and safe systems. *Systems*, 9(1), 2021.

[28] Scott Rose, Oliver Borchert, Stu Mitchell, and Sean Connelly. Zero trust architecture. Technical report, National Institute of Standards and Technology, 2020.

[29] SolarWinds. Solarwinds security advisory. Available at https://www.solarwinds.com/securityadvisory.

[30] Danielle Stewart, Jing Janet Liu, Michael W Whalen, Darren Cofer, and Michael Peterson. Safety annex for the architecture analysis and design language. In *10th European Conference Embedded Real Time Systems ERTS*, 2020.

[31] OSATE team. OSATE – Open Source AADL Tool Environment. https://osate.org. Accessed: May 20, 2023.

[32] VERDICT team. VERDICT – Verification Evidence and Resilient Design in Anticipation of Cybersecurity Threats. https://github.com/ge-high-assurance/VERDICT. Accessed: May 20, 2023.

[33] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in WPA2. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2017.