# Using statistical model checking for cybersecurity analysis[♦]

Carlos E. Budde ✉ [0000−0001−8807−1548]

Security, University of Trento, Italy
carlosesteban.budde@unitn.it

**Abstract** This work discusses an approach to estimate the likelihood of occurrence and evolution in time of software security issues. First, software vulnerability assessment is revised under the light of recent studies. Then, guidelines are proposed that allow for (formal) modelling stochastic aspects of cybersecurity-relevant scenarios. This opens a connection to the field of formal methods, where automated tools like statistical model checkers can estimate the value of property queries characterising such scenarios. But exploitable vulnerabilities and attacks in cybersecurity are rare events, which calls for specialised tools. In view of this, the work finalises presenting FIG, a statistical model checker specialised on rare event simulation. FIG, an open source software tool freely available at https://git.cs.famaf.unc.edu.ar/dsg/fig, can be used to estimate the probability of an attack within the next release cycle.

**Keywords:** Statistical model checking; Formal models for safety and security; Cybersecurity analysis; Rare event simulation.

## 1 Introduction

Taken-for-granted technologies in today's digital societies include personal healthcare appliances, assisted-driving cars, AIs that start to chat too well, nightly-build continuous integration/continuous deliveries, etc. Distributed data storage with a *central access point* stands among these feats. In fact, from the digital technologies mentioned, cloud storage is arguably the one with the biggest immediate impact in our everyday lives. Having your account stolen in social media apps is a twenty-first century bonfire story, not to mention the more serious implications of this happening with accounts that contain e.g. your banking data.

These modern dreadful scenarios often portray cybersecurity as the guardian angel that will keep our data safe at all costs. Unfortunately, black and white hats alike have long ago realised that the central access point, that makes cloud

[♦]Funded by the EU under GA n.101067199 (*ProSVED*). Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or The European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

technologies so useful for everyone, is often what also makes them prone to attacks. So much so that cryptography, which is the discipline that studies how to use a user-private password to turn our data into undecipherable mangled bits and back[†], is the most broadly taught subjects across cybersecurity curricula in European countries today—see e.g. recent studies by Dragoni et al. [16].

### 1.1 Uneducated mob

The above could sound encouraging. However, cryptography is but one of the many topics that cast a protective mail between our data and its attackers. Interestingly, it is not the weakest link in that mail—rather the opposite. The bigger picture shows several further disciplines which must function in synchrony, in order to deny unauthorised parties access to sensitive digital information.

For instance, [16, Fig. 4] shows the ten best-covered disciplines by European M.Sc. programs. However, a complete education must also include topics in *Component Procurement*, *Deployment and Maintenance*, *Distributed System Architectures*, compliance to *Policies*, etc. Current higher education and professional training cover these last examples only marginally [16, 19].

Perhaps a specially worrying case is human security, also known as the human factor, which is intimately connected to cybersecurity as it concerns how attacks to persons—rather than to machinery, code, or protocols—can compromise the security of individuals and even entire corporations. Human security includes disciplines such as *Identity Management*, *Compliance to Policies and Norms*, and (defence against) *Social Engineering*. And even though these aspects are evidently crucial to keep our data safe—it matters little if our PCs have an AES chip, when a colleague can shoulder-surf us with a latte macchiato—it has been known for over a decade that education is falling behind in these topics [19, 31].

As a result, good practices for access and human security which used to be suggestions, such as strong password keys and multi-factor authentication, are now downright enforced into (rather than expected from) users. Unfortunately, not every practice can be enforced, since high-security and ease-of-use typically stand on opposite sides of the user-experience spectrum [34]. Therefore, enforceable user-sided security has limited reach at best, so system administrators (and companies in general) must operate under the assumption that the users will not do the cybersecurity heavy-lifting.

### 1.2 Fifty shades of cybersecurity

The picture appears dire. "*But*"—we tell ourselves—"*this will not happen to* me*, whose security hygiene can put the CIA's to shame.*" Things are, of course, not that simple, with a complexity that soars as world-wide distributed servers (that store our data) run on heterogeneous hardware and firmware.

[†]This is rather *encryption*, a sub-field of cryptography. For a more precise (and serious) understanding of cryptography we refer the interested reader to e.g. [4, chs. 1 and 3].

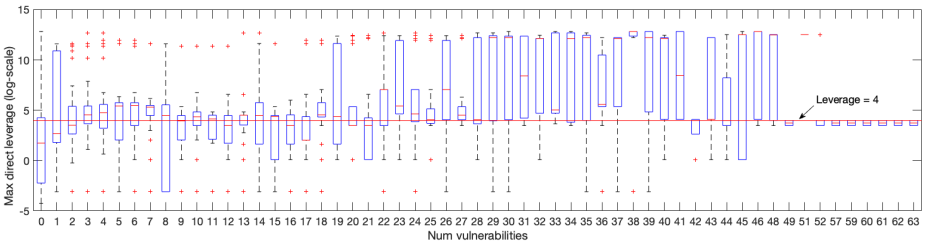


Figure 1: Higher volume of library dependencies in the owned code (higher max direct leverage in the $y$-axis) is correlated with higher number of vulnerabilities faced in the lifetime of a software project [28].

This poses a challenge to backend developers, namely efficiency (and intelligible documentation, one would hope). In turn, frontend developers are in charge of improving user experience, by offering everyone an homogeneous and equally-responsive interface. Cybersecurity appears here as a third layer, whose developers strive to achieve a common minimal *degree of protection*. And maximising this minimum is crucial, since attackers accessing data in the server are effectively bypassing any user-sided cryptographic barrier.

Ideally, system design can achieve good and homogeneous protection in a manner resilient to software heterogeneity and evolution, even in the face of undocumented "efficient" code. Unfortunately, developers seldom have the chance to design a system from scratch. More often than not companies must work with legacy software, so the choices available to developers—already constrained w.r.t. enforceable user-side security—are restricted even more. In fact, it is not uncommon that the only feasible choice is which new library version to update to [33]. The impact in security is apparent, as recent studies show correlations between the use of libraries and the number of security vulnerabilities affecting a software project—see Fig. 1 and related works [28, 32, 35].

These are practical reasons why cybersecurity cannot be Boolean, e.g. a social media platform is not either cyber-safe or -unsafe. Any software is ultimately susceptible to cyberattacks, which motivates concepts like *cyber resilience*: the ability to continuously deliver a service/product despite adverse cyber events, or to quickly and fully recover from such events [28, 33, 38].

Therefore, for companies it boils down to investment strategies: how to spend resources in a manner that reduces the risk of cyberharm, viz. the degree of protection mentioned above. Such investments range from buying specialised hardware and software, or implementing security policies, up to training attack-response teams. The ultimate goal is to lower the risk of enduring or recovering from impactful attacks: the more vulnerable the system, the higher the investment needed. Together with the increasing pressure by (inter-) national regulatory treaties—GDPR, HIPPA, PCI DSS, etc.—this has turned the estimation of software vulnerability into an increasingly hot research topic [28, 36].

Technically, however, vulnerability estimation comes with many theoretical and practical hurdles. One of the hardest to overcome is the sheer unpredictability of future technologies. In fact, most endeavours take an ostrich approach here and focus on known vulnerabilities, typically zero-day attacks, avoiding to speculate on issues to come. This line of action is mainly chosen due to the complexity of the field, where a seemingly innocuous code fragment can be exploited, but only when accessed via a specific browser with certain plugin installed.

In the face of such complexity, it is extremely difficult to determine from existent code which fragments can become vulnerable in the future, or even how many vulnerabilities one should expect later at project-level (but see [39, 45] for time-series approaches that count vulnerabilities in code-agnostic manners).

> This work discusses how an abstraction step can be taken to analyse a formalised model of the system's security. This can be used to estimate, with an arbitrary degree of accuracy, the likelihood and time lapse between the exploitation of code-specific vulnerabilities.

**Outline.** The rest of this paper is structured as follows. First, § 2 briefly revises related work. Then, a minimal introduction to model checking and its statistical variant are given in §§ 3 and 4, showing why the latter is a feasible approach to estimate (future) security vulnerabilities. However, simulation-based approaches are computationally inefficient when studying rare events, which is the category in which exploited vulnerabilities fall into. That is why § 5 presents `FIG`: a statistical model checker specialised in rare event simulation, that offers a unique solution to the estimation-by-simulation problem. This work concludes in § 6.

## 2 Related work

Studies to find software vulnerabilities date at least from the early 2000's. Most works propose a statistical approach, often demonstrating its capabilities in concrete and complex case studies or large datasets, depending on the specific objective. This has been changing from classic statistical analyses—e.g. confidence interval comparison and hypothesis tests—to modern Machine Learning (ML) approaches, ranging from concrete algorithms (support vector machines, random forests, linear regression, etc.) to full disciplines like deep learning.

For example, works like [3, 13, 29, 30, 42] study code and code-activity metrics to either correlate them to reported vulnerabilities, or identify vulnerable code fragments (i.e. the prime suspects of cybersecurity police). The features studied to find these correlations and suspicious code include average length of functions, cyclomatic complexity, nano-patterns, commit code churn, number and seniority of committers, peer comments in code and in commits, etc.

The degree of success of these approaches is variable. While most achieve relatively high sensitivity, precision is a different matter due to the high rate of false positives [2, 26, 43]. Not to mention the difficulty of extrapolation to

different projects and languages—see [35] for a discussion, and [23] for the alternative solution of anti-fragile systems. Interestingly, [35] also reports that in their study of 450 projects written in Java, Python, and Ruby, "*There is no clear relationship between dependency vulnerability count with attributes of the commit including author experience*".

Data mining and ML approaches have been surveyed in [21]. Recent works like [1, 7, 12, 20, 27] try to predict vulnerabilities* not only from software metrics but also e.g. the dependency tree, the abstract syntax tree (of function call), code cloning, etc. The main disadvantage of these approaches is the black-box models they produce: despite their accuracy, it can be hard for developers to understand why a library dependency is deemed vulnerable. In contrast, white-box models and indicators such as [26, 28, 33, 35] are better in communicating the source of the issue, which helps both developers and managers in making decisions, e.g. which dependencies to adopt, avoid, or simply update—see also [23].

All the works mentioned above attempt to detect vulnerabilities that exist in the code. Instead, the current work proposes a way to generate models that can be used to *foretell vulnerabilities that may occur in the future*. This is in line with the objectives of [39, 45], which use time-series analysis akin to those employed in the stock market—e.g. ARIMA—to estimate the number of security issues that a project may eventually face. In that sense, [39, 45] suffer from the same critique than ML: the models produced are black-box and cannot explain why is a code vulnerable, or which fix to apply. In contrast, using a formal approach as discussed in § 4 would produce white-box (formal) models that can identify the dependencies from which the estimated vulnerabilities emerge.

## 3 Model checking and cybersecurity

Formal system modelling and analysis is based on mathematical specifications of systems, whose properties are queried using (typically) temporal logic formulae. The field is vast, and a large part of it deals with *model checking* due to its attractive push-button approach, where formal checks can be fully automated [5].

### 3.1 Model checking

The fundamental steps for model checking are:

1. defining a model $M$ that describes the system to be analysed;
2. defining a property $\varphi$ that describes the query to perform;
3. checking whether (or the degree to which) the model satisfies the property, which is typically denoted $M \models \varphi$.

*In ML, *predict* refers to classification, i.e. identify code affected by a CVE or known vulnerable pattern. This is not the same as *foretelling* the occurrence of vulnerabilities in the future, e.g. a yet-to-come CVE. Here we are interested in estimating the latter.

The subsequent formal guarantees on the automatically computed answer have resulted in many success stories of model checking applied to safety analysis—a trend that continues to this day [18, 24, 37].

For example, in [37], the question from step 3 above is "*what is the probability that the power supply noise of my NoC surpasses the safety threshold*". It is compelling to see the resemblance of such queries to measuring the degree of (cyber-) security resilience. This has in fact started to be noticed, as researchers begin to apply model checking for general cybersecurity studies [17, 41].

But there is a zeroth step that precedes modelling and is often assumed:

> Selecting the formalism in which $M$ and $\varphi$ will be given semantics is crucial, because it determines the type of questions that can be asked.

For example, the semantics on which the model $M$ is interpreted must be able to speak about the passing of time (formally, allow for a continuous state space) if one desires to ask about the duration of events. Also, and quite to the point of attack-resilience, the chosen formalism must allow for probability measures, to query about the likelihood of an event taking place. We now discuss these matters for cybersecurity analysis.

### 3.2 Semantic basis

In automata theory, many mathematical formalisms can express either time or probabilities [22]. Arbitrary combinations of these aspects are less common, in part because the complexity of the resulting models quickly reaches undecidability even for reachability properties. That is, if the semantics are chosen to be too expressive, there may be no algorithm that can compute e.g. whether a vulnerable situation is reachable. Computational efficiency is also a factor to consider: the more flexible the model, the more computation steps (and runtime) it will take for an algorithm to find the answer to a query.

Simply put, the modelling formalism must be chosen as expressive as needed—to answer all relevant questions—and as simple as possible.

> For software vulnerabilities we are interested in two types of questions:
> - "*what is the probability of an attack in a defined time window?*",
> - "*what is the expected time between independent attacks?*".

Both questions are stochastic in nature, and the second one requires the estimation of potentially continuous time intervals. The simplest formalism from the literature that can cope with both continuous probability measures and time are *Stochastic Automata*, a subset of STA [14, 22].

> A Stochastic Automaton $M$ can encode the occurrence of attacks, according to probability distributions fitted from empirical data coming from real-world measurements. Then, a PCTL-like property $\varphi$ can query the time-bounded probability of observing relevant events in the foreseeable future.

This regards steps 1 and 2 from the model checking procedure. However, step 3 encounters the extra requirement of verifying *arbitrary* distributions, which come from approximations of the empirical attack information and probabilities observed in the real world. Such typically non-Markovian behaviour rules out traditional (probabilistic) model checking, whose numerical approximation algorithms—e.g. value iteration and its variants—rely on the memoryless property. Workarounds like using face-types to approximate the empirical distributions have short reach, since they increase the number of states to visit, in an already NP-hard problem of known exponential size.

Instead, the analysis of non-Markovian stochastic systems is typically approached with Monte Carlo simulation. Embedded in a formal methods setting this is usually called statistical model checking (SMC [46]).

## 4 Statistical model checking for cybersecurity

SMC integrates Monte Carlo simulation with formal methods. Via discrete event simulation it generates traces, which are samples of the states that a stochastic model $M$ can visit. Via the generation and analysis of these stochastic samples, SMC estimates the degree to which $M$ satisfies different properties.

### 4.1 Monte Carlo simulation: an informal primer

Resorting to SMC brings an extra semantic requirement: the model must be fully stochastic and hence be free of nondeterministic behaviour. The following gives an intuition of what this means; formal treatments are e.g. in [5, 15, 22].

For our purposes it suffices to consider the model $M$ as consisting of a (possibly infinite) set of states $S = \{s_0, s_1, s_2, \ldots\}$, and transitions $s_i \to s_j$ among them. Each state represents a general configuration of the modelled system, so for instance if we are studying how our server processes calls via its web API, $s_0$ can represent the idle state, $s_1$ the reception of a message via method `foo()`, $s_2$ the process of stripping the header from the payload, etc. The transitions in the model represent how the system passes from one state to another; in our example this could be $s_0 \to s_1 \to s_2$.

A simulation trace in the model $M$ is as a sequence of states $\sigma = s_{i_0} s_{i_1} s_{i_2} \cdots$ where for any two consecutive states $s_{i_j} s_{i_{j+1}}$ there exists a transition $s_{i_j} \to s_{i_{j+1}}$ in the model $M$. The trace $\sigma$ thus models "a run in the system", describing how it evolves from certain initial state onwards. For simplicity we let $s_0$ be the initial state, but one could also choose a set of initial states $S_0 \subsetneq S$.

Roughly speaking, a *simulation of Monte Carlo* is the process of generating a trace $\sigma$. This is done several times, generating a collection of traces $\{\sigma_j\}_{j=i}^{m}$ that model how the system usually operates. Each trace can then be tested against the property, e.g. $\varphi =$ "sensitive data has been leaked", to see whether the trace makes $\varphi$ true. This results in a statistical answer to $M \models \varphi$, e.g. "our server leaks data on average 0.0021% of the time, with a standard deviation of 5.2E-4".

Note however that the transitions in our model are not necessarily deterministic. For example, imagine that our web API has a `bar()` method, and let $s_{42}$ represent the reception of a message via `bar()`. This is modelled by the transition $s_0 \to s_{42}$ in $M$, which means that from the idle state there are at least two possible choices: $s_0 \to s_1$ (`foo`) vs. $s_0 \to s_{42}$ (`bar`). This is called *branching*.

> To be able to use SMC, all branching in the model must be probabilistic.

In the example above this means that there must be probabilistic weights in the transitions from $s_0$ to either of its two possible successors. In general, these probabilities need not be discrete: for continuous state spaces one would use stochastic distributions, as Stochastic Automata do. However, the generic definition of Stochastic Automata is not fully probabilistic [14]. Its representation of stochastic time periods via clocks allows nondeterministic behaviour, viz. where branching arises for which there is no quantification whatsoever of which path to follow. Then Monte Carlo cannot generate traces, because when faced with a nondeterministic choice it does not know how to continue.

But there is a subset of Stochastic Automata known to be modular and fully probabilistic: *Input/Output Stochastic Automata*, and its weakly-nondeterministic variant with so-called *urgent actions* (IOSA [15]). This means that IOSA models allow arbitrary distributions, and permit the formal analysis of properties via SMC, to estimate statistical quantities such as the probability of leaking sensitive data via web API calls.

> IOSA models have all desired properties to study cybersecurity via SMC.

### 4.2 Modelling considerations and guidelines

Regarding the design of models for cybersecurity, it is up to the modeller to decide which kind of information is encoded in the states $S$. The specific choice depends on the behaviour of interest—some straightforward options are: (a) the first-level libraries that a main project depends on, (b) the number of known vulnerabilities for the own codebase and also for these libraries, (c) the criticality of these vulnerabilities, (d) the time since their publication, and (e) whether any of these codebases is currently under attack.

In turn, there are many interfaces to represent the states and transitions of $M$ for (statistical) model checking. To avoid reinventing the wheel, any suitable pre-existent *modelling syntax* must be tried first: cybersecurity is no exception, so syntaxes that can model security-relevant situations should be explored.

Attack trees (ATs) are a simple example [44]: they are structural decompositions of the steps needed to perpetrate a (security) attack, and can be used for cybersecurity. ATs can, for instance, model option (a) above, under the assumption that a security vulnerability in a dependency can also compromise the code that uses it. Technically, the root of the tree would be an `OR` gate that represents

```
toplevel "MainLib";
"MainLib" or "own_code" "depend_1" "depend_2";
"own_code" fail~weibull(k=2.3,β=125);
"depend_1" fail~exponential(λ=8.07E-3);
"depend_2" fail~lognormal(μ=30,σ=90);
```

Code 1: Kepler AT for library & dependencies

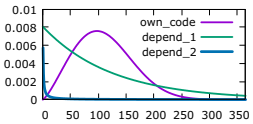


Figure 2: Attack PDFs

the main library, and the leaves would be `BAS`es with relevant information, e.g. the empirical probability distribution of vulnerability disclosure.

There are modern syntaxes to describe (fault and) attack trees, for instance Kepler [11], that can be re-interpreted to model cybersecurity scenarios. Code 1 shows one such AT for a library with two dependencies, whose hypothesised fitted PDFs for vulnerability disclosure are shown in Fig. 2.

Code 1 proposes a way to declare models for option (a) above. In contrast, option (e) is speculative and defines the goal of the simulations. More precisely, a temporal logic property $\varphi$ can query the probability of transitioning from the current safe state, to a state in which one or more of the codebases is under attack, before $T$ days have passed. SMC can estimate this value (the step $M \models \varphi$) by generating several samples via Monte Carlo simulation, and computing the proportion of them that suffered an attack before $T$ days.[‡] Here, $\varphi$ is said to characterise a subset of the states $S_\varphi \subset S$, whose reachability we are estimating.

Thus, from $M$ and $\varphi$, an SMC analysis yields an estimate $\hat{\gamma} \in [0, 1]$ of the actual probability $\gamma$ with which the model satisfies $\varphi$, e.g. the likelihood of an attack. Besides producing $\hat{\gamma}$, SMC can quantify the statistical error incurred via two numbers, $\delta \in (0, 1)$ and $\varepsilon > 0$, such that $\hat{\gamma} \in [\gamma - \varepsilon, \gamma + \varepsilon]$ with probability $\delta$. Thus, if $n \in \mathbb{N}$ traces are sampled, the full SMC outcome is the tuple $(n, \hat{\gamma}, \delta, \varepsilon)$.

This statistical quantification is usually returned as a confidence interval (CI) around $\hat{\gamma}$, and conveys an idea of the quality of the estimation. The usual approach is to fix the confidence $\delta$ prior to experimentation: then higher quality means smaller $\varepsilon$ and thus a narrower CI, achieved by drawing more samples.

### 4.3 Computational considerations

Although flexible and automatic, the SMC approach is hindered by rare events. That is, if there is a very low probability $\gamma$ to satisfy $\varphi$, then most traces sampled by SMC will not visit $S_\varphi$. The result is then either an incorrect estimate $\hat{\gamma} = 0$ or, if a few traces do visit $S_\varphi$, the confidence interval computed is very wide and hence uninformative.

This can affect cybersecurity analyses, since the likelihood of observing an exploit of a vulnerability is quite low in practice. To counter such phenomena, the number of samples $n$ must increase as $\gamma$ decreases. Unfortunately, for the sample mean, this causes $n$ to increase in inverse proportion to the square of $\gamma$,

[‡] Transitions among states are governed by stochastic distributions, that describe the jump probabilities from past evidence. Stochastic Automata encode this via "clocks".

which quickly results in unacceptably-long run times. To tackle this issue, rare event simulation (RES) methods have emerged in many scientific disciplines [40].

### 4.4 Rare Event Simulation

Roughly speaking, RES can be divided in importance sampling and importance splitting (ISPLIT). The former modifies the stochastic transitions of the model, in a way that can later be undone when computing the estimate $\hat{\gamma}$. This is not clearly feasible for cybersecurity, where the transition distributions are arbitrary as they come from empirical data.

In contrast, ISPLIT methods are not directly affected by such matters, which makes them more attractive to our purposes. A caveat is that ISPLIT traditionally requires expert knowledge to split the state space $S$ of the model $M$. This has limited the use of SMC+ISPLIT as an automatic approach in general, and specifically for cybersecurity analysis, since it necessitates (specialised) user input beyond the definition of $M$.

However, novel theories are emerging to finally automate this step [6, 8]. Next and to conclude this work, a statistical model checker that implements automatic RES is briefly presented.

## 5 The FIG tool

### 5.1 Main characteristics

The Finite Improbability Generator, FIG, is an SMC tool publicly available at https://git.cs.famaf.unc.edu.ar/dsg/fig. It uses the formal definitions of $M$ and $\varphi$ to derive the so-called importance function $f$ and thresholds $\{\ell_i\}_{i=1}^{M}$ [9], which act as an oracle guiding simulation traces towards the rare event in a tractable manner, i.e. keeping a measure of the bias introduced[¶]. These are the core components needed by ISPLIT to speed up the statistical convergence for the computation of the estimate $\hat{\gamma}$ [25].

For this, FIG runs a breadth-first search from $S_\varphi$ on the inverted transitions of $M$. This computes the number-of-transitions distance from each state to $S_\varphi$. The heuristic importance function of FIG, $f^\star$, is the inverse of this distance, stored as an array of size $|S|$. To avoid the state explosion, FIG works on IOSA modules, deriving a local $f_i^\star$ for each $M_i$ whose parallel composition forms $M$ [15]. $f^\star$ is an aggregation of these functions, which in its most basic form adds the local $f_i^\star$ of every $M_i$ whose variables appear explicitly in $\varphi$ [8, 10].

Function $f^\star$ is solely based on the distance measured in number of transitions of $M$. All stochastic behaviour that is omitted by $f^\star$, such as probabilistic weights in the transitions, is captured in the thresholds $\ell_i$. To choose these thresholds automatically, FIG runs dynamic analyses using either Expected Success or a

[¶]A more in-depth introduction to the concept of importance function requires to formally define state spaces, nondeterministic vs. probabilistic branching, and simulation traces in formal models—we refer the interested reader to e.g. [25, 8].

```
1  module Foo
2      recv : clock;           // receive a call to foo()
3      proc : clock;           // process a call to foo()
4      comm : [0..2];          // communicate with backend
5      busy : [0..2] init 0;   // processing call
6      [rc!] busy==0 @ recv -> 0.1: (recv'=μ)   // package lost
7                            + 0.9: (comm'=1) & (busy'=busy+1);
8      [c??] busy==1        -> (busy'=2) & (proc'=ν); // callback
9      [pc!] busy==2 @ proc -> (comm'=2) & (busy'=0) & (recv'=μ);
10     [b!!] comm==1        -> (comm'=0);  // communicate: busy!!
11     [d!!] comm==2        -> (comm'=0);  // communicate: done!!
12 endmodule
```

Code 2: IOSA module of `foo()` web API call for `FIG 1.3`

variant of the Sequential Monte Carlo algorithm [10]. In both cases finite-life simulations start from the initial state, to estimate roughly the probability to reach states with higher importance via lightweight statistical analyses.

### 5.2 IOSA models

FIG is designed to run either crude Monte Carlo or RES simulations on IOSA with urgency [15]. We (briefly) revise the syntax to define these models, as a reference to future cybersecurity implementations—for further details see [9].

**Modular composition.** An IOSA model is composed of one or more synchronising modules, which contain local continuous random variables called clocks. Every clock samples a positive value according to its probability distribution. As time evolves, the clocks in all modules count down at the same rate, and the first to reach zero is said to expire. On expiration a clock can trigger (a) local events in its active module—e.g. new sampling of clock values, variables assignment—and (b) synchronisations with other passive modules. The single active module whose clock expired broadcasts an output action, that synchronises with homonymous input actions in the passive modules. IOSA is an input-enabled formalism.

**Example.** Code 2 shows a simplistic FIG module that represents the reception of a message via the `foo()` method of a web API. It consists of variable declarations (lines 2–5) that define its possible internal states, and transitions (lines 6–11) that define the transitions between these states.

Generally speaking, the different situations in which the model can be (waiting for a message to arrive, expecting the callback, processing the message, etc.) are modelled by the different combination of values of the variables `comm` and `busy`. Instead, the clocks `recv` and `proc` govern the transition between these possible states, i.e. the changes in the values of the variables. This is said to happen "stochastically" by sampling their next expiration times and expiring as described above. The next paragraph gives a sample walk-through of the resulting behaviour for the IOSA model from Code 2.

A simulation starts at state `comm==0,busy==0` and, informally speaking, waits some time until a message is received. This is governed by the expiration of the clock `recv` in lines 6–7, represented by the code "`@ recv`". When this happens,

the code to the right of the arrow `->` in line 6 represents a 10% change to lose the message (e.g. due to data corruption), in which case we will repeat the process by waiting until the next message is received. With the remaining 90% chance the model will instead process the package: first it broadcasts a signal (`b!!` in line 10) to indicate that it will be busy for a while; then it waits for the backend to inform that the message is to start being processed (`c??` in line 8). This then makes the model wait for the expiration of clock `proc` (line 9), which indicates the end of the processing. When this happens, a signal is broadcast to indicate that we are done (`d!!` in line 11), after which the full procedure starts over.

Next we give a more technical explanation of the parts that compose the model from Code 2, and any IOSA model that can be simulated by `FIG`

**State variables.** IOSA variables in `FIG` have module-local scope and can be of type `clock`, `bool`, or ranged integer (e.g. `[0..2]`). Constants can also be of type `float`—though not `clock`—and have global-scope. `FIG` supports array variables and can compute e.g. a-random or the-smallest value from an array.

**Transitions.** Besides the declaration of state variables, an IOSA module is composed of transitions that describe its behaviour. Each transition is formed by: a (possibly empty) action in square brackets; a Boolean precondition; a clock iff the action is (non-urgent) output; and a postcondition, formed of a probabilistic choice (after `->` and separated by `+`) where single options have probability 1.

**Synchronisation actions.** Every transition starts with a (possibly empty) action declaration. Decorators `?`/`!` at the end of the action name mark it as input or output, e.g. `rc!` in line 6 is an output action. Double decorators are for urgency, e.g. `c??` in line 8 is an urgent input. Output actions (timed or urgent) are emitted by the module, and listened to by other IOSA modules in the system. Input actions (timed or urgent) are received by the module, when another module in the system emits them.

**Clock expiration.** Time passes at the same speed for all IOSA modules. Each clock samples a value from its corresponding distribution, e.g. $\mu$ for `recv` in Code 2, and counts down (at the global speed of time passage) until its value reaches zero: then the clock *expires*. Character `@` in a transition represents this event. If the clock of a transition expires, and the precondition is satisfied, the postcondition of the transition is applied. For instance, `[rc!] busy==0 @ recv-> ···` (line 6) tells that if clock `recv` expires, and `busy==0`, then the module will output action `rc` and proceed to execute the postcondition that follows the arrow `->`.

**Postconditions.** A postcondition is composed of one or more *options*, each weighed with a discrete probability value. Each option, chosen probabilistically, is a sequence of *effects* concatenated by `&`. The left-hand side of `=` in an effect is a variable name with a `'` suffix. If the variable is a clock, the right-hand side is a distribution: e.g. `recv'=`$\mu$ in line 6 samples a new clock value. Else, the right-hand side is an arithmetic expression: e.g. in line 7 the value `1` is assigned to variable `comm`, and `busy+1` is assigned as new value to variable `busy`.

### 5.3 Demonstration

Finally, we show the command-line interface (CLI) and capabilities of FIG to study rare-event properties in two small examples from its test suite.

FIG offers many options to simulate on IOSA models, for the estimation of rare properties specified in subsets of continuous-time stochastic logic (CSL, for steady-state properties) or probabilistic computation tree logic (PCTL, for transient properties). A full description of this CLI is out of scope, but users can get it by invoking the `--help` option of FIG, after downloading it from https://git.cs.famaf.unc.edu.ar/dsg/fig and following the installation instructions.

The first example that we will showcase is a triple tandem queue with Erlang service times: the IOSA model file is publicly available in the official website of FIG in the following path: `tests/models/3tandem_queue.sa`.

We compare crude Monte Carlo (CMC) and two RES strategies with the monolithic importance function, i.e. $f^\star$ built on the composition of all IOSA modules. The first strategy uses all of FIG default parameters, and the second one requests Expected Success to build thresholds, and the RESTART engine with level-2 prolongations. The corresponding commands are:

```
>_ fig --stop-time 5m 3tandem_queue.sa --cmc
   fig --stop-time 5m 3tandem_queue.sa --amono
   fig --stop-time 5m 3tandem_queue.sa --amono -t es -e restart2
```

We estimated the CSL-like property $\varphi =$ `S(q3>=7)`, which asks the proportion of time that the third queue contains more than 7 elements. Comparisons were done for a fixed simulation budged, namely a wall-clock time of 5 minutes. When the time is due, simulations stop and CIs are reported: the estimation that achieves the narrowest CI for a fixed confidence level is the most efficient one.

Running these experiments in an Intel(R) Xeon(R) E-2124G CPU @ 3.40GHz (Linux kernel 5.14.8-arch1-1) resulted in the following 95% CIs: [3.81E-6, 4.52E-6] for CMC, [4.15E-6, 4.36E-6] for FIG defaults, and [4.25E-6, 4.40E-6] for the custom command. The widths of these intervals are 7.13E-7, 2.12E-7, and 1.53E-7 resp.

All CIs overlap and contain the expected value 4.25E-6. However and as expected, RES can achieve tighter estimates for the same simulation budget. We highlight that the default FIG command is as bare as crude Monte Carlo, yet it produced an estimate more than three times more precise.

Finally we experiment with a second model: a small repairable Fault Tree with non-Markovian failure and repair times (`FT.sa`), also available in the website of FIG as `tests/models/resampling_tiny_FT.sa`. The distribution families include exponential, Erlang, normal, and lognormal.

The case is quite interesting since ISPLIT has limited applications in FT analysis. Importance functions such as $f^\star$, that only observe failures and repairs of components, result in efficient RES applications iff the dominant failure can be layered, e.g. as the result of the conjunctive failure of many subcomponents. To exploit this, we have developed heuristics that automatically derive a composition strategy from the FT structure. *A similar approach is envisioned for cybersecurity studies, using the closely related theory of attack tree analysis.*

For this case, we estimated the time-bounded probability of observing a system failure before 150 time units. Again we compare CMC and two RES strategies: `FIG` with the `--ft` switch, Expected Success thresholds, Fixed Effort simulation engine, and (a) the default compositional importance function, and (b) the heuristic FT-structure importance function. The commands are:

```
>_ fig --stop-time 5m FT.sa --cmc
   fig --stop-time 5m FT.sa --ft -t es -e sfe --acomp +
   fig --stop-time 5m FT.sa --ft -t es -e sfe --acomp \
       'BE_0+max(BE_1,BE_2)+BE_4'
```

These experiments resulted in the 95% CIs: [1.93E-4, 3.02E-4], [2.28E-4, 3.12E-4], and [2.39E-4, 2.70E-4], whose widths are 1.09E-4, 8.41E-5, and 3.12E-5. As before, all CIs contain the expected value (2.65E-4), and RES achieved the tightest intervals for the same simulation budget. In this case, however, the difference between CMC and the default compositional strategy of `FIG` is much less pronounced than in the previous example. This is expected given the low redundancy required to cause a system failure (three components must be simultaneously failed).

Yet in spite of this, the heuristic composition strategy performed significantly better, producing a CI almost an order of magnitude narrower than CMC. Perhaps the most appealing feature of this strategy is that it is automatic: it is computed from the FT structure, from which also the IOSA modules were created. In other words, this is effectively a fully-automatic deployment of RES. In subsequent research we intend to apply analogous approaches to study properties of models that encode cybersecurity problems.

## 6 Conclusions

This work discussed the use of Statistical Model Checking to study problems relevant for cybersecurity practices. SMC is a formal approach to model analysis via Monte Carlo simulation. Input/Output Stochastic Automata semantics are proposed as underlying formalism: they combine continuous time and probabilities, as required to estimate the likelihood and time of occurrence of future attacks. The need for rare event simulation is identified to achieve efficient computations, and the academic tool `FIG` is presented, which can deploy it automatically.

## References

1. Akram, J., Luo, P.: SQVDT: A scalable quantitative vulnerability detection technique for source code security assessment. Software: Practice and Experience **51**(2), 294–318 (2021). https://doi.org/10.1002/spe.2905
2. Alohaly, M., Takabi, H.: When do changes induce software vulnerabilities? In: CIC. pp. 59–66. IEEE (2017). https://doi.org/10.1109/CIC.2017.00020
3. Alves, H., Fonseca, B., Antunes, N.: Software metrics and security vulnerabilities: Dataset and exploratory study. In: EDCC. pp. 37–44. IEEE (2016). https://doi.org/10.1109/EDCC.2016.34

4. Aumasson, J.P.: Serious Cryptography: A Practical Introduction to Modern Encryption. No Starch Press (2017)
5. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
6. Barbot, B., Haddad, S., Picaronny, C.: Coupling and importance sampling for statistical model checking. In: TACAS. LNCS, vol. 7214, pp. 331–346. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_23
7. Bilgin, Z., Ersoy, M.A., Soykan, E.U., Tomur, E., Çomak, P., Karaçay, L.: Vulnerability prediction from source code using machine learning. IEEE Access **8**, 150672–150684 (2020). https://doi.org/10.1109/ACCESS.2020.3016774
8. Budde, C.E.: Automation of Importance Splitting Techniques for Rare Event Simulation. Ph.D. thesis, Universidad Nacional de Córdoba, Córdoba, Argentina (2017)
9. Budde, C.E.: FIG: The Finite Improbability Generator v1.3. SIGMETRICS Perform. Eval. Rev. **49**(4), 59–64 (2022). https://doi.org/10.1145/3543146.3543160
10. Budde, C.E., D'Argenio, P.R., Hartmanns, A.: Automated compositional importance splitting. Science of Computer Programming **174**, 90–108 (2019). https://doi.org/10.1016/j.scico.2019.01.006
11. Budde, C.E., D'Argenio, P.R., Monti, R.E., Stoelinga, M.: Analysis of non-Markovian repairable fault trees through rare event simulation. Int. J. Softw. Tools Technol. Transfer **(to appear)** (2022). https://doi.org/10.1007/s10009-022-00675-x
12. Chakraborty, S., Krishna, R., Ding, Y., Ray, B.: Deep learning based vulnerability detection: Are we there yet. IEEE Transactions on Software Engineering **48**(9), 3280–3296 (2021). https://doi.org/10.1109/TSE.2021.3087402
13. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. Journal of Systems Architecture **57**(3), 294–313 (2011). https://doi.org/10.1016/j.sysarc.2010.06.003
14. D'Argenio, P.R., Katoen, J.P.: A theory of stochastic systems part I: Stochastic automata. Inf. Comput. **203**(1), 1–38 (2005). https://doi.org/10.1016/j.ic.2005.07.001
15. D'Argenio, P.R., Monti, R.E.: Input/Output Stochastic Automata with Urgency: Confluence and weak determinism. In: ICTAC. LNCS, vol. 11187, pp. 132–152. Springer (2018). https://doi.org/10.1007/978-3-030-02508-3_8
16. Dragoni, N., Lafuente, A.L., Massacci, F., Schlichtkrull, A.: Are we preparing students to build security in? A survey of European cybersecurity in higher education programs [education]. IEEE Security & Privacy **19**(01), 81–88 (2021). https://doi.org/10.1109/MSEC.2020.3037446
17. Fang, Z., Fu, H., Gu, T., Qian, Z., Jaeger, T., Hu, P., Mohapatra, P.: A model checking-based security analysis framework for IoT systems. High-Confidence Computing **1**(1) (2021). https://doi.org/10.1016/j.hcc.2021.100004
18. Faqeh, R., Fetzer, C., Hermanns, H., Hoffmann, J., Klauck, M., Köhl, M.A., Steinmetz, M., Weidenbach, C.: Towards dynamic dependable systems through evidence-based continuous certification. In: ISoLA. LNCS, vol. 12477, pp. 416–439. Springer (2020). https://doi.org/10.1007/978-3-030-61470-6_25
19. Furnell, S., Clarke, N.: Power to the people? the evolving recognition of human aspects of security. Computers & Security **31**(8), 983–988 (2012). https://doi.org/10.1016/j.cose.2012.08.004
20. Ganesh, S., Ohlsson, T., Palma, F.: Predicting security vulnerabilities using source code metrics. In: SweDS. pp. 1–7. IEEE (2021). https://doi.org/10.1109/SweDS53855.2021.9638301

21. Ghaffarian, S.M., Shahriari, H.R.: Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Comput. Surv. **50**(4) (2017). https://doi.org/10.1145/3092566
22. Hartmanns, A.: On the analysis of stochastic timed systems. Ph.D. thesis, Saarland University (2015). https://doi.org/10.22028/D291-26597
23. Hole, K.J.: Anti-fragile ICT Systems. Springer (2016). https://doi.org/10.1007/978-3-319-30070-2
24. Khan, S., Katoen, J.P.: Synergising reliability modelling languages: BDMPs and repairable DFTs. In: PRDC. pp. 113–122. IEEE (2021). https://doi.org/10.1109/PRDC53464.2021.00023
25. L'Ecuyer, P., Le Gland, F., Lezaud, P., Tuffin, B.: Splitting techniques. pp. 39–61. Wiley (2009). https://doi.org/10.1002/9780470745403.ch3
26. Li, H., Kwon, H., Kwon, J., Lee, H.: A scalable approach for vulnerability discovery based on security patches. In: ATIS. CCIS, vol. 490, pp. 109–122. Springer (2014). https://doi.org/10.1007/978-3-662-45670-5_11
27. Li, Q., Song, J., Tan, D., Wang, H., Liu, J.: PDGraph: A large-scale empirical study on project dependency of security vulnerabilities. In: DSN. pp. 161–173. IEEE (2021). https://doi.org/10.1109/DSN48987.2021.00031
28. Massacci, F., Pashchenko, I.: Technical leverage in a software ecosystem: Development opportunities and security risks. In: ICSE. pp. 1386–1397. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00125
29. Meneely, A., Williams, L.: Secure open source collaboration: An empirical study of linus' law. In: CCS. pp. 453—462. ACM (2009). https://doi.org/10.1145/1653662.1653717
30. Meneely, A., Williams, L.: Strengthening the empirical analysis of the relationship between Linus' law and software security. In: ESEM. ACM (2010). https://doi.org/10.1145/1852786.1852798
31. Parsons, K., Calic, D., Pattinson, M., Butavicius, M., McCormac, A., Zwaans, T.: The human aspects of information security questionnaire (HAIS-Q): Two further validation studies. Computers & Security **66**, 40–51 (2017). https://doi.org/10.1016/j.cose.2017.01.004
32. Pashchenko, I., Plate, H., Ponta, S.E., Sabetta, A., Massacci, F.: Vulnerable open source dependencies: counting those that matter. In: ESEM. pp. 42:1–42:10. ACM (2018). https://doi.org/10.1145/3239235.3268920
33. Pashchenko, I., Plate, H., Ponta, S.E., Sabetta, A., Massacci, F.: Vuln4Real: A methodology for counting actually vulnerable dependencies. IEEE Transactions on Software Engineering **48**(5), 1592–1609 (2022). https://doi.org/10.1109/TSE.2020.3025443
34. Post, G.V., Kagan, A.: Evaluating information security tradeoffs: Restricting access can interfere with user tasks. Computers & Security **26**(3), 229–237 (2007). https://doi.org/10.1016/j.cose.2006.10.004
35. Prana, G.A.A., Sharma, A., Shar, L.K., Foo, D., Santosa, A.E., Sharma, A., Lo, D.: Out of sight, out of mind? How vulnerable dependencies affect open-source projects. Empirical Software Engineering **26**(4) (2021). https://doi.org/10.1007/s10664-021-09959-3
36. Rindell, K., Ruohonen, J., Holvitie, J., Hyrynsalmi, S., Leppänen, V.: Security in agile software development: A practitioner survey. Information and Software Technology **131** (2021). https://doi.org/10.1016/j.infsof.2020.106488
37. Roberts, R., Lewis, B., Hartmanns, A., Basu, P., Roy, S., Chakraborty, K., Zhang, Z.: Probabilistic verification for reliability of a two-by-two network-on-chip sys-

tem. In: FMICS. LNCS, vol. 12863, pp. 232–248. Springer International Publishing (2021). https://doi.org/10.1007/978-3-030-85248-1_16
38. Rose, A.Z., Miller, N.: Measurement of Cyber Resilience from an Economic Perspective, chap. 10, pp. 253–274. John Wiley & Sons, Ltd (2021). https://doi.org/10.1002/9781119287490.ch10
39. Roumani, Y., Nwankpa, J.K., Roumani, Y.F.: Time series modeling of vulnerabilities. Computers & Security **51**, 32–40 (2015). https://doi.org/10.1016/j.cose.2015.03.003
40. Rubino, G., Tuffin, B.: Introduction to rare event simulation. pp. 1–13. Wiley (2009). https://doi.org/10.1002/9780470745403.ch1
41. Stoelinga, M., Kolb, C., Nicoletti, S.M., Budde, C.E., Hahn, E.M.: The marriage between safety and cybersecurity: Still practicing. In: SPIN. LNCS, vol. 12864, pp. 3–21. Springer (2021). https://doi.org/10.1007/978-3-030-84629-9_1
42. Sultana, K.Z., Deo, A., Williams, B.J.: Correlation analysis among Java nanopatterns and software vulnerabilities. In: HASE. pp. 69–76. IEEE (2017). https://doi.org/10.1109/HASE.2017.18
43. Walden, J., Stuckman, J., Scandariato, R.: Predicting vulnerable components: Software metrics vs text mining. In: ISSRE. pp. 23–33. IEEE (2014). https://doi.org/10.1109/ISSRE.2014.32
44. Weiss, J.: A system security engineering process. In: Proceedings of the 14$^{th}$ National Computer Security Conference. Information System Security: Requirements & Practices, vol. 249, pp. 572–581 (1991)
45. Yasasin, E., Prester, J., Wagner, G., Schryen, G.: Forecasting IT security vulnerabilities – an empirical analysis. Computers & Security **88** (2020). https://doi.org/10.1016/j.cose.2019.101610
46. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: CAV. LNCS, vol. 2404, pp. 223–235. Springer (2002). https://doi.org/10.1007/3-540-45657-0_17