BADGraph: A Tool for Quantitative Modeling and Analysis of Probabilistic Attack Scenarios

Double-blind submission

Abstract. BADGraph is a novel tool for the quantitative modeling and analysis of highly customizable threat scenarios. A BADGraph model consists of an attributed attack-defense diagram to describe vulnerabilities, defenses and countermeasures, and the probabilistic behaviour of an attacker acting on the diagram, possibly constrained by limited resources and attack effectiveness. BADGraph offers a domain specific language, with formal semantics, through a modern integrated development environment. BADGraph models are analyzed using statistical model checking, allowing to study properties ranging from average cost and success probability of attacks, to the effectiveness of defenses.

1 Introduction

Graph-based security models, like variants of attack trees [9], allow to intuitively represent and formally analyze vulnerabilities in complex modern IT systems. Several approaches based on attack trees exist (see, e.g., [5]). Our own approach is BADGraph (Behavioral Attacker Defender Graphs), a novel tool available at `http://bit.ly/BADGraph` for the modeling and analysis of graphical security models. The modeling part combines *declarative* attributed attack-defense diagrams with *procedural* probabilisitic behaviour of attackers acting on, and implicitly constrained by, the diagrams. BADGraph diagrams combine common features from established approaches, such as countermeasures [10], attack detection rates [2], ordered attacks [4,8], and further common features from vulnerability analysis such as attack effectiveness. The main novelty of our approach is its unique, to the best of our knowledge, dual declarative-procedural nature, by which the combination of diagrams with attacker behavior allows to study the vulnerabilities of a system for specific classes of attackers.

BADGraph can be seen as a recast to the security domain of a recent approach proposed for software product line engineering [3,12]. Indeed, that approach was indirectly applied to a confined class of security scenarios [3], but with limitations that require intermediate encoding into security notions. Due to these limitations, in our proposal, we redevelop BADGraph from scratch following the main intuitions from [3,12], tailored for security scenarios.

BADGraph offers a high-level domain specific language (DSL) equipped with formal semantics. The formal machinery is hidden to the user by a multi-platform one-click-install IDE for the DSL. Thanks to tool-chaining with the statistical analyser MultiVeStA [11], BADGraph can study properties like the success probability and average cost of attacks, or the effectiveness of defenses.

Outline Sect. 2 discusses BADGraph's architecture. Sect. 3 and 4 illustrate the tool by applying it to a bank robbery scenario. Sect. 5 concludes the paper.

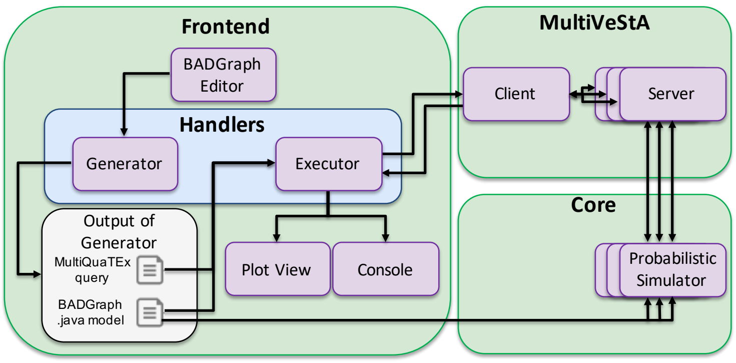


Fig. 1: BADGraph architecture and control flow

2 BADGraph Architecture

BADGraph has three major components, as shown in Fig. 1: a frontend based on Xtext (`https://www.eclipse.org/Xtext/`); a core containing a probabilistic simulator; and MultiVeStA [11], a simulation-based statistical analysis engine.

The frontend consists of a modern IDE including the BADGraph editor (providing state-of-the-art editing support like auto-completion and syntax/error highlighting), a plot view to show analysis results, and a console to show diagnostic information. The frontend also offers a set of handlers. The most noticeable ones are the generator, and the executor. The generator is triggered every time a BADGraph model is saved. In particular, the user specifies both the model and the properties of interest using a high-level DSL, and the generator takes care of generating two low-level files: a `.java` representation of the model used by the core to perform probabilistic simulations; and a query in MultiVeStA's property specification language.

The executor takes care of the analysis phase. It is responsible for instantiating MultiVeStA and to provide it with the generated files. The executor also takes care of visualizing in the plot view the analysis results.

Analysis is performed thanks to a tool-chaining with the statistical model checker MultiVeStA by means of several probabilistic simulations. Simulations can be distributed across the cores of the used machine thanks to the client-server architecture of MultiVeStA. The MultiVeStA client iteratively distributes batches of simulations on the available servers until achieving the desired statistical confidence. The parallelism degree, the size of the batches, and the level of statistical confidence are parameters specified by the modeler.

The core mainly consists of a probabilistic simulator for BADGraph models. Intuitively, probabilistic simulations are obtained by executing the simulator step-by-step starting from the initial configuration, at each step selecting one of the possible next-states according to probabilities computed from the model. The core is integrated with the MultiVeStA servers, such that each server has its own instance of the simulator.

3 Modeling a Bank Robbery Scenario

This section presents a simple bank robbery case study to showcase several modeling capabilities of BADGraph. The full BADGraph specification is available at http://bit.ly/BADGraphModelsFASE20.

The BADGraph diagram is depicted in Fig. 2. Red circles denote *attack nodes*, i.e., (sub-)goals to be accomplished for an attack to succeed. The goal is to rob a bank, the root `RobBank`, which can be achieved by opening the vault (`OpenVault`) *or* blowing it up (`BlowUp`). In order to open the vault, the attacker needs to get to the vault (`GetToVault`) *and* learn its combo (`LearnCombo`). BADGraph also supports *ordered and* relations, meaning that, e.g., one might impose that opening the vault would require the sub-attacks to succeed in a given order. For security reasons, the vault has three opening codes (`FindCode1-3`), of which at least two are required by the 2:3 relation, an instance of a *k-out-of-n* relation. Instead, `BlowUp` only requires to get to the vault, which is therefore a shared node. There are two defensive mechanisms. First, a *countermeasure* `LockDown` (purple diamond), which can be triggered by (successful or not) `BlowUp` attacks (dotted blue arrow), and, once active, it affects `RobBank` attacks (dashed gray line). The rationale is that if an explosion is detected, the vault gets sealed preventing the robbery. Second, a *defense* `Memo` (green box) is permanently active against attacks aiming to find code 2. Defenses can be disabled by attack nodes: if the attacker uses a `LaserCutter`, s/he can break `LockDown`. BADGraph diagrams are specified in a textual DSL, which is ommitted since it follows closely the visual representation in Fig. 2. An accompanying paper will present in greater detail BADGraph's syntax and semantics.

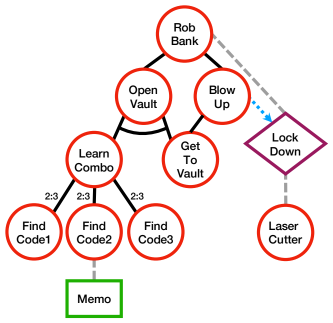


Fig. 2: BADGraph diagram

```
begin defense effectiveness
 Memo(ALL, FindCode2) = 0.5
 LockDown(ALL, RobBank) = 0.3
end defense effectiveness

begin attack detection rates
 BlowUp = 1.0
end attack detection rates

begin attributes
 Cost = {LaserCutter = 10,
  BlowUp = 90, FindCode1 = 5,
  FindCode2 = 5, FindCode3 = 5}
end attributes

begin quantitative constraints
 value(Cost) <= 100
end quantitative constraints
```

Code 1: Further properties

Diagrams can be decorated with further properties, as shown in Code 1. In particular, defensive mechanisms decrease the success rates of given attacks, specified in terms of `defense effectiveness`. Code 1 states that `Memo` scales the probability of succeeding in `FindCode2` attacks by $1 - 0.5$, while `LockDown` scales that of `RobBank` by $1 - 0.3$. `LockDown` gets activated if `BlowUp` attempts are detected. The probability of detecting an attack is given in terms of its `detection rate` (in Code 1 we state that `BlowUp` attempts are always detected). Attack nodes can be equipped with quantitative attributes. In Code 1 we have costs for attempting attacks. Code 1 shows that attributes can be used in `quantitative constraints`, e.g., to constrain the resources of attackers.

BADGraph allows to define probabilistic behaviour of attackers which aim at exploiting the vulnerabilities described in a BADGraph diagram. Fig. 3 sketches an attacker, named `Thief`, which `starts` choosing to attempt open vault (`tryOpen`) or blow up (`tryBlow`) attacks strategies. Independently from this choice, s/he can also try get-to-vault attacks (`tryGetTo`), required by both strategies. `OpenVault` requires `LearnCombo` attacks, attempted in `tryLearn`, which in turn require finding at least two codes, attempted in `tryFind`. For the sake of clarity, we do not detail the `tryOpen` strategy in Fig. 3. We can see that transitions can have sets of labels, two of which are mandatory: the performed action, and a weight used to probabilistically choose the transition to be executed. Actions can be user-defined, like `choose`, and `gtv`, or the predefined `add` and `fail`, which model successful or failed attacks, respectively. Every time an `add` or `fail` action is executed, the cost of the attempted attack node is added to an internal counter maintaining the cumulative cost of the attacker. Transitions can also have *guards*, like `allowed`, used to attempt `RobBank` in `start` only if all required sub-attacks succeeded. Another example is `!has`, used in Fig. 3 to forbid the transition towards `tryGetTo` if one already succeeded in `GetToVault`.

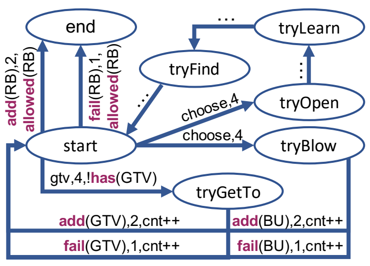


Fig. 3: Sketch of behaviour

```
begin action constraints
 do(choose) ->
  !(has(OpenVault) or has(BlowUp))
end action constraints
```

Code 2: Action constraints

```
begin init
 Thief = {FindCode1}
end init
```

Code 3: Initial configuration

BADGraph also supports *action guards*. These constrain any transition containing a given action. As shown in Code 2, any transition with action `choose` is disabled as soon as one succeeds in opening or blowing up the vault. This is because one needs to succeed only in one attack strategy at a time.

Transitions can also be labeled with side-effects, like `cnt++` in those at the bottom of Fig. 3, which updates the value of the *variable* `cnt` upon the execution of the transition. Variables can be freely defined and used in constraints or in the analysis phase. In our model, `cnt` counts the number of attempted attacks.

A BADGraph model is completed with an initial configuration, shown in Code 3, which specifies the attacker and its initially accomplished attacks (if any). In Code 3 we have that the attacker already obtained the first code.

4 Analyzing a Bank Robbery Scenario

BADGraph supports quantitative analysis of probabilistic attack scenarios by means of statistical model checking (SMC) [1,6,7]. This is obtained thanks to the integration with MultiVeStA [11], a framework for enriching simulators with SMC capabilities. We opted for SMC because the BADGraph language has high expressivity, allowing for potentially unbounded variables and high variability in the models, thus often giving rise to large or even infinite state spaces.

We showcase the analysis capabilities of BADGraph by studying the probabilities of activating attacks and countermeasures at the varying of the simulation step. This is expressed in Code 4: `from-to-by` specifies that we are interested

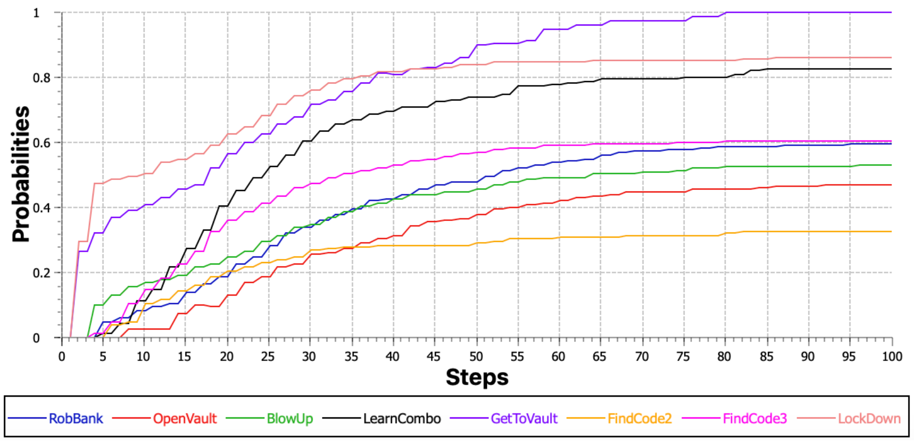


Fig. 4: Analysis result of the properties in Code 4

in the first 100 steps of simulation. Then we list the 8 properties of interest (one per attack, considering `FindCode1` is always active, plus the countermeasure `LockDown`). Each property can be an arithmetic expression of nodes (evaluating to 1 or 0 if the node is active or not, respectively), variables, or attributes. The 8 properties are considered in each of the 100 steps, for a total of 800 properties.

```
begin analysis
 query = eval from 1 to 100 by 1 : { RobBank ,
  OpenVault , BlowUp , LearnCombo , GetToVault ,
  FindCode2 , FindCode3 , LockDown }
 default alpha = 0.1 delta = 0.1 parallelism = 1
end analysis
```

Code 4: Analysis of the scenario

Each such actual property p_i denotes a random variable X_i which gets a real value assigned in each simulation. MultiVeStA estimates the expected value $E[X_i]$ of each of the 800 properties (reusing the same simulations) as the mean $\overline{x}_i$ of n independent simulations, with n large enough to guarantee an (α, δ) *confidence interval* (CI): $E[x_i]$ belongs to $[\overline{x}_i - {}^{\delta}\!/2, \overline{x}_i + {}^{\delta}\!/2]$ with statistical confidence $(1 - \alpha) \cdot 100\%$. The CI is given by `default alpha` and `delta` (but a property-specific δ could be used instead). Finally, `parallelism` states how many local processes should be launched to distribute the simulations. Overall the analysis required 400 simulations, performed in 16 seconds on a standard laptop machine.

Fig. 4 shows the analysis results. We remark that, by Fig. 2, `RobBank` requires `OpenVault` or `BlowUp`. The probability of activating `RobBank` reaches about 0.6 after 100 steps, while those of `OpenVault` and `BlowUp` reach 0.45 and 0.55, respectively. Given that by Code 2 `OpenVault` and `BlowUp` cannot both be activated, one should be able to activate `RobBank` with probability almost 1 after 100 steps. Instead, the actual probability is scaled down by the probabilistic choice done from `start` to `end` in Fig. 3, where `RobBank` can either succeed or fail.

We note that `LockDown` has a high probability of getting activated, as it reaches about 0.85 after 60 steps. This is coherent with Code 1, stating that any `BlowUp` attempt is detected. One might expect that the probabilty of having

`BlowUp` active should be higher than that of `LockDown`. However, this is not the case, as it reaches about 0.53 after 80 steps. This is explained by the fact that both succeeded and failed `BlowUp` attempts are detected, and in Fig. 3 we have both success (`add(BU)`) and failure (`fail(BU)`). Interestingly, if we add `LockDown` to the initial configuration in Code 3, then the probability of activating `LockDown` remains 0, as it is inhibited by `LockDown`.

5 Conclusion

We presented BADGraph, a quantitative modeling and verification environment for probabilistic attack scenarios, including Eclipse-based tool support. BADGraph features a modern IDE for specifying possible attacks to a system or an organization, and probabilistic behavior of attackers. This is expressed in terms of a high-level domain specific language, making the underlying use of formal methods transparent to the user. BADGraph also offers advanced statistical analyses of attacker-specific non-functional properties.

References

1. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comp. Simul. **28**(1), 6:1–6:39 (2018)
2. Amenaza: The SecuITree® BurgleHouse Tutorial (a.k.a., Who wants to be a Cat Burglar?) (2006), https://www.amenaza.com/downloads/docs/Tutorial.pdf
3. ter Beek, M.H., Legay, A., Lluch Lafuente, A., Vandin, A.: A framework for quantitative modeling and analysis of highly (re)configurable systems. IEEE Trans. Softw. Eng. (2018)
4. Çamtepe, S.A., Yener, B.: Modeling and detection of complex attacks. In: Proceedings of SecureComm'07. pp. 234–243. IEEE (2007)
5. Kordy, B., Piètre-Cambacédès, L., Schweitzer, P.: DAG-based attack and defense modeling: Don't miss the forest for the attack trees. Comput. Sci. Rev. **13–14**, 1–38 (2014)
6. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: Proceedings of RV'10. LNCS, vol. 6418, pp. 122–135 (2010)
7. Legay, A., Lukina, A., Traonouez, L., Yang, J., Smolka, S.A., Grosu, R.: Statistical Model Checking. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science: State of the Art and Perspectives, LNCS, vol. 10000, pp. 478–504 (2019)
8. Lv, W., Li, W.: Space based information system security risk evaluation based on improved attack trees. In: Proceedings of MINES'11. pp. 480–483. IEEE (2011)
9. Mauw, S., Oostdijk, M.: Foundations of attack trees. In: Won, D.H., Kim, S. (eds.) Proceedings of ICISC'05. LNCS, vol. 3935, pp. 186–198 (2005)
10. Roy, A., Kim, D.S., Trivedi, K.S.: Attack countermeasure trees (ACT): towards unifying the constructs of attack and defense trees. Secur. Commun. Netw. **5**(8), 929–943 (2012)
11. Sebastio, S., Vandin, A.: MultiVeStA: Statistical model checking for discrete event simulators. In: Proceedings of ValueTools'13. pp. 310–315. ACM (2013)
12. Vandin, A., ter Beek, M.H., Legay, A., Lluch Lafuente, A.: QFLan: A tool for the quantitative analysis of highly reconfigurable systems. In: Proceedings of FM'18. LNCS, vol. 10951, pp. 329–337 (2018)

TOOL DEMONSTRATION

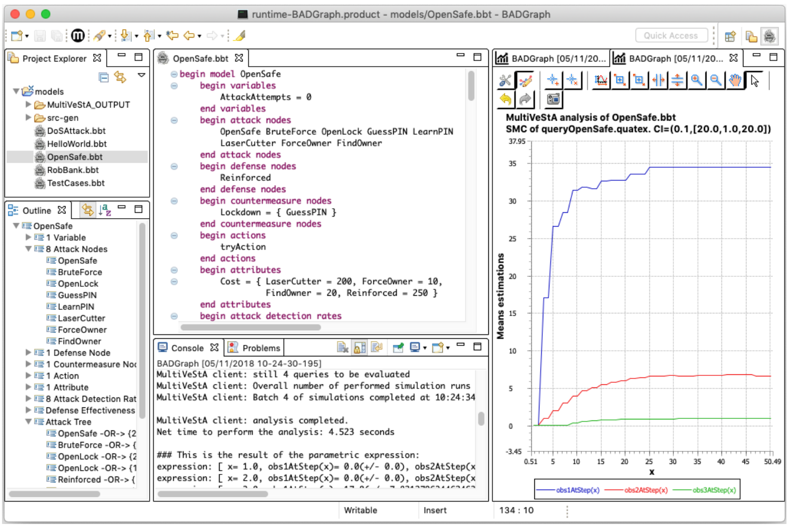
Double-blind submission

The demonstration will be organised in two parts of 6 and 9 minutes, resp.:

A (6 minutes) Presentation of the BADGraph tool;
B (9 minutes) Modeling and analysis in BADGraph of a classic example.

A The BADGraph tool

In the first part we will present the tool's functionalities using the bank robbery scenario from the main text. This part of the demonstration will be based on the material presented in the main text. We will first present the architecture of BADGraph, focusing on its GUI (shown in Fig. 1), and then we will show how to use the tool to model, modify, and analyze the bank robbery scenario.



```
begin model OpenSafe
    begin variables
        AttackAttempts = 0
    end variables
    begin attack nodes
        OpenSafe BruteForce OpenLock GuessPIN LearnPIN
        LaserCutter ForceOwner FindOwner
    end attack nodes
    begin defense nodes
        Reinforced
    end defense nodes
    begin countermeasure nodes
        Lockdown = { GuessPIN }
    end countermeasure nodes
    begin actions
        tryAction
    end actions
    begin attributes
        Cost = { LaserCutter = 200, ForceOwner = 10,
                 FindOwner = 20, Reinforced = 250 }
    end attributes
    begin attack detection rates
```

```
BADGraph [05/11/2018 10-24-30-195]
MultiVeStA client: still 4 queries to be evaluated
MultiVeStA client: Overall number of performed simulation runs
MultiVeStA client: Batch 4 of simulations completed at 10:24:34

MultiVeStA client: analysis completed.
Net time to perform the analysis: 4.523 seconds

### This is the result of the parametric expression:
expression: [ x= 1.0, obs1AtStep(x)= 0.0(+/- 0.0), obs2AtStep(x
expression: [ x= 2.0, obs1AtStep(x)= 0.0(+/- 0.0), obs2AtStep(x
```

Fig. 1: The BADGraph tool (available at http://bit.ly/BADGraph)

B Schneier's safe lock scenario

In the second part we will consider a classic case study. We will use a BADGraph model based on the safe lock scenario from Schneier's seminal work on attack trees [1], extended to better illustrate our approach. In the rest of this text we present in detail the considered model, together with the performed analyses. Instead, during the demonstration we will remain at a higher-level by presenting the models using figures only, and by discussing the performed analyses using the plots in this text. The full BADGraph specification is available at http://bit.ly/BADGraphModelsFASE20.

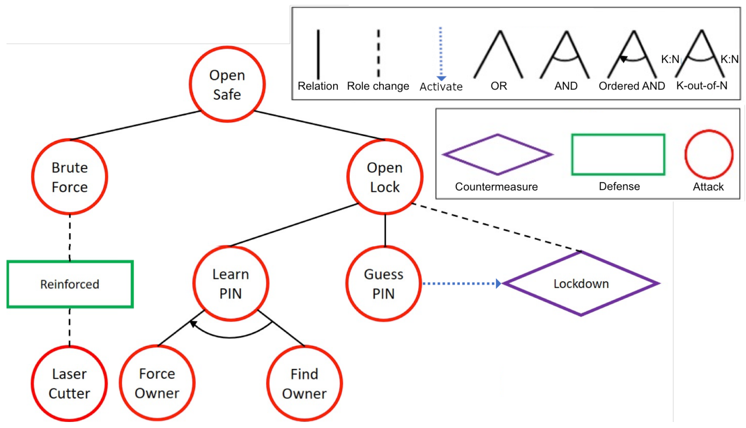


Fig. 2: BADGraph diagram

Attack-defense diagram. The attack-defense diagram of the scenario is given in Fig. 2. The goal is to open a safe, the root `OpenSafe`, which can be achieved by `BruteForce` or `OpenLock` attacks. The latter can be in turn obtained via `LearnPIN` or `GuessPIN` attacks. `LearnPIN` requires two conditions, in a given order: first to find the owner of the PIN, and then to force her/him to reveal it. There are two defensive mechanisms: a *countermeasure* `Lockdown` (the diamond), which can be triggered by `GuessPIN` attacks (denoted by a dotted arrow), and, once active, it affects `OpenLock` attacks (denoted by a dashed line). Second, there is a *defense* `Reinforced` (the box), that is permanently active against `BruteForce` attacks. As discussed in the main text, defenses can be disabled by attack nodes: if the attacker uses a `LaserCutter`, s/he can break `Reinforced`.

Attacker behaviour. The *attacker behaviour* of this scenario is sketched in Fig. 3. It has three possible strategies: `stratGP`, `stratBF`, and `stratLP`, which involve `GuessPIN`, `BruteForce`, and `LearnPIN` attacks, respectively. The first, `stratGP`, tries the following attacks: `GuessPIN` (`stratGP_GP`), `OpenLock` (`stratGP_OL`), and `Open-Safe`(`stratGP_OS`). Fig. 3 omits side effects and guards. The model has three user-defined actions, discussed later. We recall that an attacker behaviour is *executed* as follows: at each step we consider the outgoing transitions from the current state of the behaviour which are admitted by the attack diagram and by the other constraints. Among those, we use their relative weights to probabilistically choose one (e.g., from `start` to `stratGP` with probability $\frac{1}{1+2+10}$).

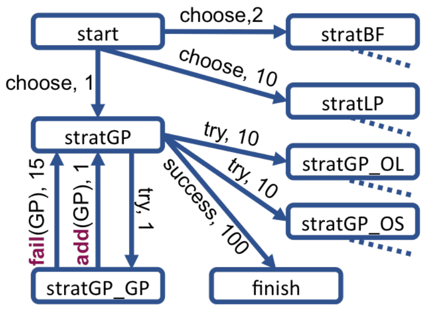


Fig. 3: Attacker behaviour

B.1 Actual BADGraph model

Nodes, Variables and Actions. Code 1 shows how the nodes of our running example are declared. We recall that when declaring a countermeasure node (`Lockdown`) it is also necessary to specify which attack nodes can trigger it (`GuessPIN`). Instead, Code 2 shows that our example has one variable, used to count the attack attempts, while Code 3 shows the user-defined actions of our example: `choose` is executed when selecting the attack strategy, `try` when attempting an attack, and `success` when the overall attack succeeded.

```
begin attack nodes
 OpenSafe BruteForce
 OpenLock GuessPIN
 LearnPIN LaserCutter
 ForceOwner FindOwner
end attack nodes
begin defense nodes
 Reinforced
end defense nodes
begin countermeasure nodes
 Lockdown = { GuessPIN }
end countermeasure nodes
```

Code 1: Nodes

```
begin variables
 AttackAttempts = 0
end variables
```

Code 2: Variables

```
begin actions
 choose try
 success
end actions
```

Code 3: Actions

```
begin attributes
 Cost = {LaserCutter=200, FindOwner=20,
         ForceOwner=10, Reinforced=250}
end attributes
```

Code 4: Attributes

```
begin defense effectiveness
 Reinforced(ALL,BruteForce)=0.95
 Lockdown(ALL,OpenLock)=0.8
end defense effectiveness
```

Code 5: Defense effectiveness

```
begin attack detection rates
 BruteForce=0.5,OpenLock=0.1,GuessPIN=0.8,LearnPIN=0.3
 LaserCutter=0.6,ForceOwner=0.7,FindOwner=0.6
end attack detection rates
```

Code 6: Attack detection rates

Node properties. Code 4 shows the attribute `Cost` used in our example to denote the cost of attempting an attack . The default value is 0, e.g. `Cost(GuessPIN)`=0. Instead, the defense effectiveness in our example are given in Code 5, where `ALL` denotes any attacker. We recall that BADGraph supports attack detection rates, used to determine the probability that an attack attempt is detected, triggering the activation of the affected countermeasures. The default value is 0, i.e., the attack is not detectable. Code 6 specifies this property for our example.

Attack-defense Diagram. Lines 2-4 of Code 7 set the hierarchical constraints of our example (cf. Fig. 2): to add `OpenSafe` we must have `BruteForce` or `OpenLock`, which in turn requires `GuessPIN` or `LearnPIN`, which requires `FindOwner` and `ForceOwner`, *in this order* (i.e., squared brackets of `OAND`). Lines 5-7 of Code 7 show the role-changing relations of the attack and countermeasure nodes. The execution of `add` and `fail` actions on an attack node with a defensive child is allowed, but their success probability is decreased according to the defense effectiveness. Instead, the addition of an attack node deactivates all defenses and countermeasures having it as role-changing child, and prevents future activations of the countermeasures.

```
begin attack diagram                              1
 OpenSafe -> {BruteForce, OpenLock}               2
 OpenLock -> {GuessPIN, LearnPIN}                 3
 LearnPIN -OAND-> [FindOwner,ForceOwner]          4
 BruteForce -> {Reinforced}                       5
 Reinforced -> {LaserCutter}                      6
 OpenLock -> {Lockdown}                           7
end attack diagram                                8
```

Code 7: BADGraph diagram

```
begin quantitative constraints
 { value(Cost) < 250 }
 { AttackAttempts < 15 }
end quantitative constraints
```

Code 8: Quantitative constraints

```
begin action constraints
 do(add(LaserCutter)) -> {value(Cost) < 100}
 do(fail(LaserCutter)) -> {value(Cost) < 100}
end action constraints
```

Code 9: Action constraints

Constraints. Code 8 gives the quantitative constraints of our example: "*The accumulated cost of an attack may not exceed 250*", and "*An attacker may not attempt more than 15 attacks*". Instead, Code 9 shows the action constraints: laser cutter attacks are not allowed if the cost exceeded 100.

```
 1  begin attacker behaviour
 2   begin attack
 3    attacker = clever
 4    states = start, finish, stratForce, stratLearnPIN, stratGuessPIN, stratGP_GuessPIN,
 5      stratGP_OpenSafe, stratGP_OpenLock, stratF_OpenSafe, stratF_BruteForce,
 6      stratF_LaserCutter, stratLP_OpenSafe, stratLP_OpenLock, stratLP_LearnPIN,
 7      stratLP_ForceOwner, stratLP_FindOwner
 8    transitions =
 9     //Pick a strategy
10     start - (choose,  1) -> stratGuessPIN,
11     start - (choose,  2) -> stratForce,
12     start - (choose, 10) -> stratLearnPIN,
13
14     //Strategy GuessPIN
15     stratGuessPIN - (try, 1, allowed(GuessPIN) and !has(GuessPIN)) -> stratGP_GuessPIN,
16     stratGP_GuessPIN - (add(GuessPIN), 1, {AttackAttempts = AttackAttempts + 1}) ->
          stratGuessPIN,
17     stratGP_GuessPIN - (fail(GuessPIN), 15, {AttackAttempts = AttackAttempts + 1}) ->
          stratGuessPIN,
18
19     stratGuessPIN - (try, 10, allowed(OpenLock) and !has(OpenLock)) -> sGPOpenLock,
20     stratGP_OpenLock - (add(OpenLock), 10, {AttackAttempts = AttackAttempts + 1}) ->
          stratGuessPIN,
21     stratGP_OpenLock - (fail(OpenLock), 1, {AttackAttempts = AttackAttempts + 1}) ->
          stratGuessPIN,
22
23     stratGuessPIN - (try, 10, allowed(OpenSafe) and !has(OpenSafe)) -> stratGP_OpenSafe,
24     stratGP_OpenSafe - (add(OpenSafe), 10, {AttackAttempts = AttackAttempts + 1}) ->
          stratGuessPIN,
25     stratGP_OpenSafe - (fail(OpenSafe), 1, {AttackAttempts = AttackAttempts + 1}) ->
          stratGuessPIN,
26
27     stratGuessPIN - (success, 100, has(OpenSafe)) -> finish
28
29     //Strategies Force and LearnPIN ...
30   end attack
31  end attacker behaviour
32
33  begin init
34   clever = { FindOwner }
35  end init
```

Code 10: Attacker behaviour

Behaviour. Code 10 sketches the attacker behaviour `clever`. It has a `start` and a `finish` state, and some of intermediate ones (Lines 4-7). Lines 10-12 show that `start` can evolve in three states: `stratGuessPIN`, `stratForce`, and `stratLearnPIN`, each implementing an attack strategy involving only the corresponding attacks. As shown in Line 12, the latter strategy has the highest weight, namely 10, and hence it is chosen with higher probability. Lines 14-27 detail the simplest strategy, `stratGuessPIN`. Lines 15-17 try to add `GuessPIN`: if the constraints allow it (`allowed(GuessPIN)`), and it has not been added yet (`!has(GuessPIN)`), Line 15 executes the user-defined action `try` to move to the intermediate state `stratGP_GuessPIN`. Once there, given that it is unlikely to guess a PIN, we `add` it with weight 1, or `fail` with weight 15 (Lines 16-17), and then return to `stratGuessPIN`. Notably, the transitions with `add` and `fail` increment variable `AttackAttempts`, which hence counts the attack attempts. Lines 19-21 and Lines 23-25 are similar, but regard `OpenLock` and `OpenSafe`. Lines 33-35 complete the specification with an *initial status*: the attacker used, `clever`, and the pre-accomplished attacks, `FindOwner`.

B.2 Analysis

Analysis on first attack. We first study the probability that each attack is attempted as first and that it succeeds, as well as the average steps to attempt the first attack. Lines 1-4 of Code 11 express these nine properties (one probability per attack node, and the average steps): `query` specifies the properties to be `eval`uated in the first state `when AttackAttempts==1`.

Table 1 shows the results for the analysis on first attack (cf. Lines 1-4 of Code 11). The probability of achieving `OpenSafe`, `OpenLock`, or `LearnPIN` on first attempt is 0. This is due to the hierarchical constraints from the diagram (e.g., in order to achieve `LearnPIN` we must have `FindOwner` and `ForceOwner`). The probability of `BruteForce` and `GuessPIN` is low because the attacker is more likely to try the strategy for `LearnPIN`. Also, `BruteForce` attempts are likely to fail due to the high defense effectiveness of `Reinforced`. `FindOwner` has probability 1 because it is included in the initial configuration (Code 10). Finally, on average, 3 steps are necessary to attempt an attack. This is consistent with Code 10: first a strategy is chosen (Lines 10-12), then `try` is executed towards an intermediate state (e.g., Line 15) where an attack is attempted (e.g., Lines 16-17). The analysis required 400 simulations, ran in 1.22 seconds on a Mac with a 2.4 GHz Intel Core i5 and 4 GB of RAM.

```
begin analysis                                                                                  1
 query = eval when { AttackAttempts==1 } : { OpenSafe , BruteForce , OpenLock , GuessPIN ,      2
      LearnPIN , LaserCutter , ForceOwner , FindOwner , steps[delta=0.5] }
 default delta = 0.1  alpha = 0.05  parallelism = 1                                             3
end analysis                                                                                    4
                                                                                                5
begin analysis                                                                                  6
 query = eval from 1 to 50 by 1 : { OpenSafe , BruteForce , OpenLock , GuessPIN , LearnPIN ,    7
      LaserCutter , ForceOwner , FindOwner }
 default delta = 0.1  alpha = 0.05  parallelism = 1                                             8
end analysis                                                                                    9
```

Code 11: The two kinds of supported analysis

`OpenSafe`	`BruteForce`	`OpenLock`	`GuessPIN`	`LearnPIN`	`LaserCutter`	`ForceOwner`	`FindOwner`	`steps`
0	0.013	0	0.013	0	0.056	0.27	1	3

Table 1: Results for for the analysis on first attack (cf. Lines 1-4 of Code 11)

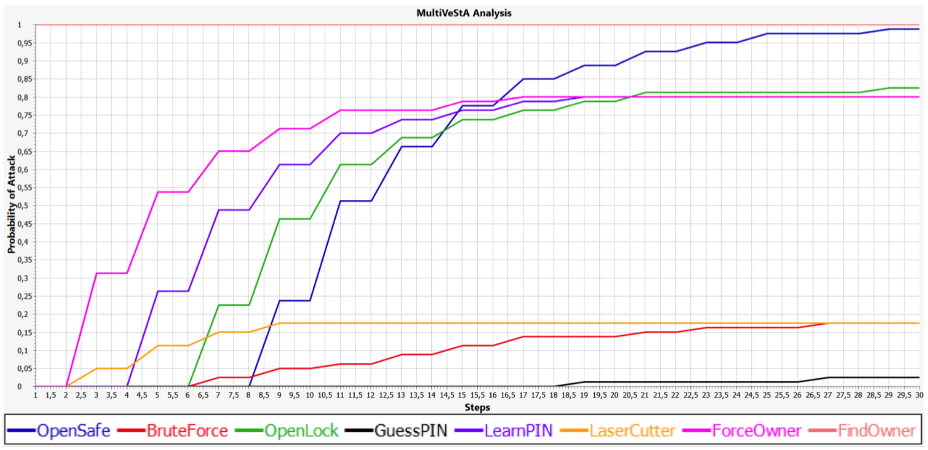


Fig. 4: Probability of successful attacks (over time, cf. Lines 6-9 of Code 11)

Analysis over time. Lines 6-9 of Code 11 study properties at the varying of the simulation step. This requires to change the condition `when` with the steps of interest: `from` 1 `to` 50, increasing `by` 1. We also removed `steps`. This means we are evaluating 50×8 properties. Fig. 4 shows the probability of achieving each attack. The one of `OpenLock` grows, until approaching 1 at step 30. Then probabilities stabilize and we truncate the plots. `GuessPIN` has the lowest probability, followed by `BruteForce` and `LaserCutter`, while the others stabilize close to 0.8.

References

1. Schneier, B.: Attack trees. Dr. Dobb's Journal (1999), https://www.schneier.com/academic/archives/1999/12/attack_trees.html