BGPy: The BGP Python Security Simulator

Justin Furuness
University of Connecticut
Storrs, CT, USA
jfuruness@gmail.com

Cameron Morris
University of Connecticut
Storrs, CT, USA
cameron.morris@uconn.edu

Reynaldo Morillo
University of Connecticut
Storrs, CT, USA
reynaldo.morillo@uconn.edu

Amir Herzberg
University of Connecticut
Storrs, CT, USA
amir.herzberg@gmail.com

Bing Wang
University of Connecticut
Storrs, CT, USA
bing@uconn.edu

ABSTRACT

The security of Border Gateway Protocol (BGP), and inter-domain routing in general, remains a challenge, in spite of its well-known importance, repeated attacks and incidents, and extensive efforts and research over decades. We present BGPy, an open-source, extensible, robust, easy-to-use and efficient BGP security simulator, to be used for research and education. BGPy allows realistic simulations of a large variety of BGP attacks and defenses. It is provided as a Python package, and can be further customized and extended, e.g., to investigate new attacks and new defense mechanisms. We describe how BGPy is currently used by multiple BGP security projects.

ACM Reference Format:
Justin Furuness, Cameron Morris, Reynaldo Morillo, Amir Herzberg, and Bing Wang. 2023. BGPy: The BGP Python Security Simulator. In *2023 Cyber Security Experimentation and Test Workshop (CSET 2023), August 07–08, 2023, Marina del Rey, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3607505.3607509

1 INTRODUCTION

BGP, the backbone protocol of the Internet's inter-domain routing system, lacks built-in authentication measures and is frequently subjected to misconfigurations and different attacks, with far-reaching consequences [4, 49, 57, 58, 60]. Accordingly, there is extensive literature on BGP's vulnerabilities, possible attacks, and defense strategies (see surveys [8, 29, 30, 42, 52] and the references within). These studies often rely on simulations for large-scale evaluation. Indeed, the Internet is of an immense scale, with over 75K autonomous systems (ASes) and 500K edges in the current AS topology [10], and its inter-domain routing is highly complex, governed by many economic and policy considerations, and the impact of BGP security mechanisms depend on adoption and policy by many different ASes. Hence, it is impractical to analytically study Internet routing and its security. Emulation-based approaches (e.g., [19]) that run the exact BGP protocol also have limitations in that they can only accommodate small topologies.

Therefore, simulations are the main tool to study BGP security. However, existing BGP security simulators lack flexibility and are hard to extend. They are in fact, mostly designed for specific attacks and defenses. In this paper, we develop *BGPy*, a Python-based simulator for studying BGP security. BGPy is designed to have the following main features:

• *Flexibility.* BGPy allows different security policies to be easily plugged in. It supports flexible configurations of attack scenarios and defense strategies. For instance, it allows multiple-AS attacks and real-world attacks (e.g., as reported in [6, 9, 39]). It supports not only the common scenario of partial deployment of a security policy, but also the realistic situation of mixed deployment of multiple security policies, e.g., Route Origin Validation (ROV) [17, 33, 62] and ROV++ [43], a security extension to ROV. Furthermore, BGPy can take into account known adoption of specific policies and defenses.

• *Efficiency.* BGPy supports large-scale simulation of attacks and defenses over empirically derived Internet-size topologies. It further allows a large number of simulation runs in a short amount of time to obtain statistical results.

• *Benchmark evaluation and reproducibility.* BGPy provides a set of testing scenarios and security metrics that can be used to test a wide range of security techniques to achieve repeatable and apples-to-apples comparisons.

• *Usability and availability.* BGPy makes it easy to perform simulations to study BGP security, facilitating use by the research community, and is available as open-source.

To realize the above design features, BGPy separates the actual simulation from the specification of the simulation into two components, the *Simulation Engine* and the *Simulation Framework*. The Simulation Engine achieves efficient propagation of BGP announcements over large AS topologies. It allows each AS to execute its specific policies, e.g., dropping announcements following ROV [17, 33, 62], or adding blackhole or preventive announcements following ROV++ [43]. This allows BGPy to easily support various security policies, including simulating real-world deployment, as well as partial or mixed deployment. It provides much more flexibility than algorithm-based propagation as used by [26–28, 64], which is only applicable to specific, 'built-in' security policies.

The Simulation Framework is a wrapper around the Simulation Engine. It allows for trials on various attack and defense scenarios, and obtaining important metrics for security evaluation. In addition, BGPy has a third main component, the *System Test Suite*, which is

ACM ISBN 979-8-4007-0788-9/23/08…$15.00
https://doi.org/10.1145/3607505.3607509

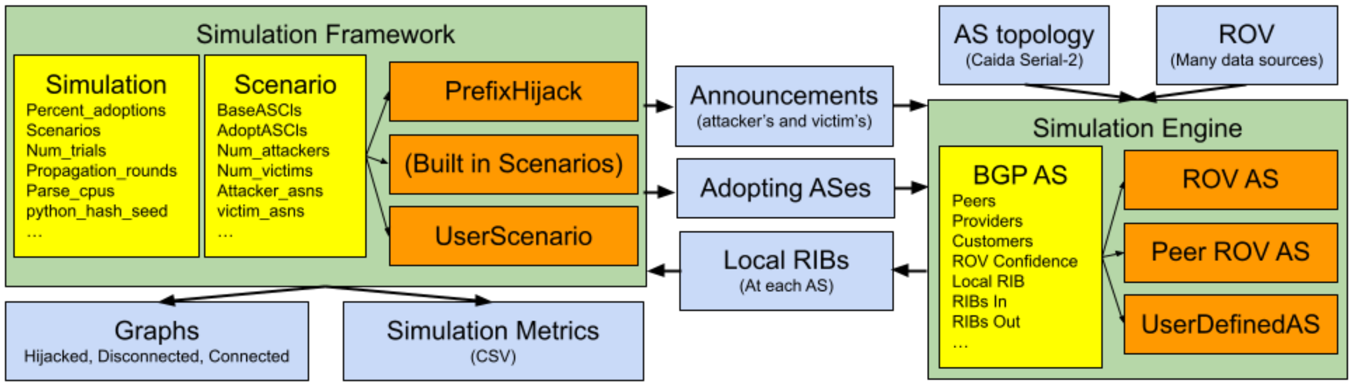

Figure 1: Main components of BGPy. Each scenario selects the attacker strategy in the form of announcements, as well as which ASes will adopt. The simulation engine receives this information and propagates these announcements throughout the AS graph, returning the LocalRIB at each AS. The simulation Framework then generates metrics and graphs.

a testing framework that provides user-friendly tools that facilitate easy modeling and debugging. BGPy is written in Python, a fast prototyping language that allows easy further extension.

BGPy has been used in several projects on BGP security (§6). These projects have demonstrated that BGPy is efficient, and can run on standard laptops/desktops without the need of computing clusters (Appendix C). In addition, it is easily extensible to support new security policies, sometimes with just a few lines of additional code. We have open sourced BGPy for facilitating BGP security research: https://github.com/jfuruness/bgpy

Contributions. The contributions of this work include:

• Design and implementation of BGPy. We design and implement BGPy as a Python-based simulation tool that aims to be efficient, flexible and easily extensible to support a wide range of attacks and security policies.

• Evaluation of BGPy. We present several use cases that use BGPy for simulating defenses against prefix hijacks, subprefix hijacks, and path manipulation, in partial and mixed deployment scenarios. Our results demonstrate that BGPy realizes its design goals, and can be a valuable tool for BGP security research.

Related work. Several studies have developed BGP simulators. Specifically, SSFNet [16, 46], Genesis [55], and BGP++ [19] are BGP simulators at the packet level. They provide fine-grained simulation, but are difficult to scale, and hence not suitable for simulating Internet-size topologies. BGPSIM [63] and Cisco WAE [54] abstracts away some details in BGP, but still has high computational overhead. The studies in [20, 21, 47] focus on the simulation at a single AS. The above simulators are mainly designed for networking research (e.g., studying convergence time and routing dynamics), not for security.

For BGP security research, it is important to consider the full Internet AS topology [26]. Existing studies used simulations that are designed for specific security policies (e.g., [15, 25–28, 32, 64]), rather than presenting general extensible simulation tools. Specifically, the studies in [26–28] use routing tree algorithms to compute the best available paths from each AS to the destination (origin of the prefix). While the algorithms are efficient, they cannot be easily extended to simulate custom security protocols. We simulate the propagation of BGP announcement inside an AS topology, where each AS can plug in a specific security policy, which allows much more flexibility, including easy support of mixed deployment of security policies. In addition, the run time complexity of our Simulation Engine is similar to that of the routing tree algorithms (see §4). The simulator in [64] is also based on algorithms, instead of actual propagation of BGP announcements, and their complexity is worse than that of BGPy.

2 BGPY DESIGN OVERVIEW

In this section we present a high level overview of BGPy. BGPy simulates BGP and allows comparison of security policies against various attacks. It consists of three main components: Simulation Engine, Simulation Framework, and System Test Suite.

Simulation Engine (§4). As shown in Figure 1, the simulation engine abstracts away packet-level and intra-domain details to perform BGP simulations by propagating announcements across the entire AS topology. The simulation engine receives an empirically inferred ASes topology and relationships from the CAIDA Serial-2 [10] dataset, announcements to populate the AS graph, and ROV adoption estimates from [14, 50, 51]. From there, the AS topology is populated, along with the routing policies defined in the BGPAS class. Among other attributes, this class contains relationship information, RIB information, and routing policies. This class is easily extendable and ROVAS (performing ROV) and PeerROVAS (performing an ROV variant) are included by default as well as subclasses. The simulation engine supports dynamic routing policies, and can use any routing policy at any AS. After these announcements are propagated throughout the AS topology, the Local RIB at each AS is produced as output (and used by the framework).

Simulation Framework (§3). As shown in Figure 1, the simulation framework is a wrapper around the Simulation Engine that facilities the comparison of multiple security policies against attack scenarios. It contains two major components - the main simulator and the scenarios. The simulator controls all scenarios and aspects of the simulation such as the number of trials, partial adoption percentages, etc. The simulator has a list of several scenarios to compare. Each scenario controls the attacking strategy, i.e. which

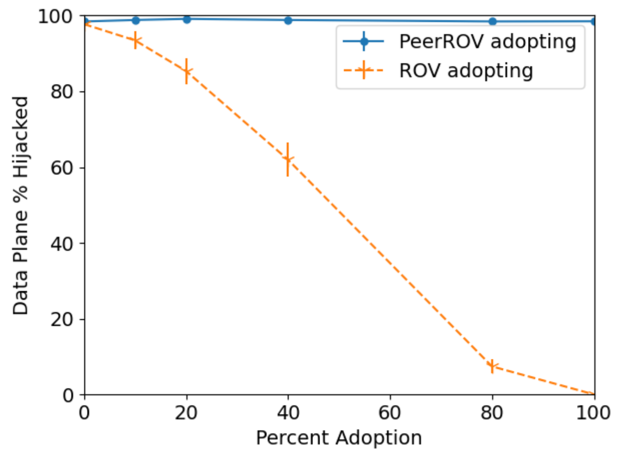

Figure 2: Example of one of the graphs displayed from the simulation Framework. Here ROV ASes are compared against an ROV variant, Peer ROV (where ROV is only applied to peers, see Appendix E), against a subprefix hijack. Percent of ASes hijacked on the data plane is measured on the Y axis.

.

ASes will be the attacker, which announcements they will send, etc. Each scenario also controls the defensive strategy, such as which ASes adopt defensive routing policies, which routing policies do they adopt, etc.

System Test Suite (§5). The final component is the system test suite. It contains useful tools for easy visualization and debugging, which is useful when working with complex AS topologies and routing policies. The tools enable the entire simulation to be converted to and from YAML at any point, and allows the simulation to be visualized in diagram form.

3 THE SIMULATOR FRAMEWORK

In this section we discuss the simulator framework, a wrapper around the simulation engine which we describe in §4.

The simulation framework sets up the simulation engine, controls all of the variables for comparing various attack defense scenarios, performs the data analysis for these scenarios, and graphs the output. We discuss all these features and more in this section. For a full view of all components related to the simulation framework, please see Figure 1.

3.1 Functionality

BGPy's primary objective is to investigate the outcome of deploying a defensive routing policy to a limited extent in response to specific attack scenarios.

All of the variables, parameters, and functions used in such an investigation are defined within a *scenario*; in particular, a scenario defines the attack strategy and the routing policies and defenses of different ASes. An attack strategy specifically consists of which announcements the attacker and victim will announce, and the defensive routing policy is the policy that will be adopted to defend against such an attack by adopting ASes across the AS topology. The scenario also controls other aspects of the simulation, such as how attackers, victims, and adopting ASes get selected, and what routing policies and defenses are used by specific ASes. The last ability is critical, as it allows to take advantage of known policies and known adoption of defenses, e.g., ASes known to adopt ROV. For more details on parameters and extensions, see §3.2.

The simulation framework allows us to compare these various scenarios under comparable conditions to let the user analyze various pros and cons of different defense policies, and directly compare them to one another in order to determine recommendations, tradeoffs, and effectiveness of security policies under specific conditions.

The following steps provide an overview of the operational flow of the simulation framework, from initiation to completion. Throughout the discussion, we will reference examples from a specific simulation that compares ROV with an ROV variant that only filters announcements from peers, PeerROV (see Appendix E for details). These security policies are used against a subprefix hijack. This simulation produces output graphs Figure 2, and we also include code snippets for multiple components.

1. Simulation Configuration: To initiate the setup of our simulation, we have the ability to adjust numerous options. For our example simulation case, which yields the depicted outcomes in Figure 2, our objective is to determine the most effective security policy against a subprefix hijack. This hijack occurs when an attacker falsely originates a subprefix of a larger IP space that is legitimately announced by another AS. We compare the effectiveness of two security policies: *standard ROV* and *PeerROV*. PeerROV is a modified version of ROV that solely filters announcements from peers, deployed by some ASes[7, 14], discussed further in Appendix E. In the provided code snippet responsible for generating the results shown in figure Figure 3, we can observe that we have specifically chosen to compare these two scenarios. Furthermore, we are interested in assessing how these security policies perform under partial adoption. Hence, in our example, we have opted to test various levels of adoption, ranging from a single AS adopting to all ASes except one adopting the policies. We have configured the number of trials to be 1000, and we have allocated 12 CPU cores, taking advantage of the fact that the trials are capable of parallelization.

```
Simulation(
    percent_adoptions = (
        SpecialPercentAdoptions.ONLY_ONE,
        .1,  # This means 10 percent of ASes adopt
        .2,
        .4,
        .8,
        SpecialPercentAdoptions.ALL_BUT_ONE
    ),
    scenarios=(
        SubprefixHijack(AdoptASCls=ROVSimpleAS),
        SubprefixHijack(AdoptASCls=PeerROVSimpleAS),
    ),
    output_path=Path("~/Desktop/cset23_graphs").expanduser(),

    num_trials=1000,
    parse_cpus=12,
).run()
```

Figure 3: Simulation Example code used to generate Figure 2 and other graphs not depicted here. This simulation compares both ROV and an ROV variant, Peer ROV (an ROV variant that only filters peers, see Appendix E for details), against a subprefix hijack. It does so for multiple partial adoption percentages, for 1000 trials, using 12 cores for multiprocessing.

All of these options are configurable. For a full list of simulator parameters, see Table 3. We could also compare any number of scenarios, and each scenario is also highly configurable (see Table 4 for a full list of parameters).

2. Attacker and Victim Selection: Before each scenario runs, the simulator randomly selects attacker and victim ASes. A *victim* is an AS that is the origin of a legitimate announcement. An *attacker* is an AS that announces an announcement that is illegitimate for any reason. Perhaps the announcement is invalid by ROA, or the announcement is a route leak, etc. The goal of the attacker(s) could be to attract, to itself, traffic sent to the victim, to cause a Denial of Service, e.g., preventing traffic from reaching the victim, or another, user-defined goal. A scenario can select any number of attackers or victims as a parameter (see Table 4); this capability was used in some use-cases (see §6). The default is one attacker and one victim.

The same attacker(s) and victim(s) are used across all scenarios in a trial to maintain comparability. By default, attackers are selected from either stubs or multihomed ASes, since edge ASes are more likely to be malicious [12, 37, 66].

3. Adopting ASes: Then the adopting ASes will be chosen. The simulator will take each scenario and run the scenario across a configurable set of different partial adoption percentages. For example, in our code used in figure Figure 3, we are testing six scenarios: one AS adopting, 10%, 20%, 40% and 80% adopting ASes, and finally, when all ASes, except one, adopt. For each of these adoption percentages, adopting ASes must be chosen. One challenge we experienced when adopting ASes is that there is a wide variety in connectivity between the ASes. A stub AS adopting will not have nearly as much impact as a highly connected AS adopting. This can result in large variance when using uniform random selection of ASes. To decrease the variance, we separate the ASes into three subcategories: ASes that are stubs or multihomed, ASes that are a part of the input clique defined by CAIDA [10], and other ASes (to which we refer as *etc ASes*). For example, if 50% of all ASes should adopt the defensive policy, 50% of the input clique ASes will be chosen, 50% of the etc ASes will be chosen, and 50% of the stubs or multihomed ASes will be chosen.

Attacker and Victim ASes are by default excluded from these selections. By default, attacker ASes are not adopting, and victim ASes are adopting, although this can easily be extended and modified.

Note that these subgroupings are configurable. A user can specify any subgroupings according to one's preferences. Similar to the attacker and the victim, the ASes chosen to be adopting ASes will remain the same across all defensive policies for each trial to enforce comparability.

4. Engine Setup: After the adopting ASes are using the routing policies chosen as the defense for the scenario, the attacking strategy is utilized to create and insert the originating announcements at the local RIBs of both the victims and the attackers. This functionality is contained within the scenario class, and creating the attacker strategy is discussed in §3.2.

The simulator includes by default six different attacking strategies, that cover the typical ‘family’ of ‘generalized prefix attacks’ that are commonly used when studying ROV and ROV++. They are defined in Table 5.

5. Engine runs: Then the simulation engine, described in §4, is run. The attackers and victims announcements are propagated throughout the AS topology, and the defensive routing policies for the adopting ASes are used. At the conclusion of this stage, the simulation engine contains an AS topology where each AS will have a local RIB containing some subset of the attacker’s and victim’s announcements.

6. Data Analysis: At this point, the framework begins to perform data analysis. To start with, we perform *traceback*, a process in which, at each AS, we trace back each prefix to it’s origin on the data plane. Figure 7, which describes a subprefix hijack originating at AS 666, shows an example for the importance of traceback. . Without traceback, i.e., looking only at the control-plane data, it would seem that AS 3 is not hijacked by AS 666, since AS 3 would be routing according to the legitimate announcement (from AS 777). However, in reality, and as the framework finds by traceback, the traffic is hijacked once it reaches AS 1 (the provider of AS 3), since AS 1 routes according to the subprefix hijack from AS 666. This is since Internet Protocol (IP) routing prefers the most-specific route. For this reason, it is not enough to simply look at the local RIB at each AS. We must perform *traceback*, tracing the prefixes back to their origins, in order to determine the true outcome of the traffic. During traceback, we start at each AS, and trace back the prefixes on the data plane to the origin AS. The outcome of the traceback is then used to analyze a multitude of metrics. The default metrics included are percent of ASes hijacked (indicating that the announcement at an individual AS was traced back to the attacker), percent of ASes disconnected (indicating that the announcement was traced back to neither the attacker nor the victim), and percent of ASes successfully connected (indicating that the announcement was traced back to the victim AS). We further divide these metrics into various subgroups, such as in Figure 2, the percent of stub or multihomed ASes that were adopting and traced back to the attacker, indicating attacker success. These metrics are easily extendable to keep track of custom metrics the user wants to track.

7. Graphs and Example Simulation From here, the resulting data is output into a CSV, and 18 default graphs are output as well. The user can also use the resulting CSV to create their own graphs as they wish. The CSV contains, for each scenario, for each percent adoption, the average tracked metrics (percent of ASes that were hijacked, disconnected, or successfully connnected). An example of these graphs can be seen in Figure 2.

3.2 Refining the Framework: Empowering Customization

The simulation framework offers a wide range of optional parameters that can be easily configured during setup, providing flexibility for customization. Moreover, its design enables straightforward subclassing and the creation of simple, personalized functions, which we will explore through various examples in the following sections.

Configurable Parameters. There are many parameters that can be set when running a simulation that can affect various aspects of the analysis. The parameters for the simulator can be seen in Table 3. While the simulator parameters affect all scenarios that a user wants to compare (for example, setting the python_hash_seed will make all scenarios deterministic), there are also configuration options for each individual scenario. For example, setting the AdoptASCls to ROV will set that specific scenario to defend using ROV, but other scenarios may have different values for AdoptASCls and different defensive strategies. The parameters for the various scenarios can be seen in Table 4. We offer a range of preconfigured attack strategies accompanied by various scenarios for a users selection (refer to Table 5). It is crucial to distinguish between the defensive and attacking strategies implemented in our system. The defensive strategy can be customized through the parameter AdoptASCls, whereas the attacking strategy necessitates the use of its dedicated subclass. As a result, the scenarios listed in the table implement their own attacking strategy in a subclassed function, while the parameters still accommodate different defensive strategies, attacker counts, and more.

Extendable Classes. There are also many functions and classes that have been written in such a way that users can easily subclass and override them to control almost any aspect of the simulations. For example, §6 lists drastically different simulations that did not need to modify the source code of the simulator and were facilitated by subclassing, made possible by BGPy's modular design. Subclassing allows basically unlimited customizations, and was extensively used in different simulations. Here we describe the three most common classes that get extended:

• **AS Class** The AS class controls the routing policy for a given AS. Within this AS class, a user can easily control decisions such as path selection and export policy. Two common functions that are often overridden in this class is the function used to rank announcements (called '_new_ann_better', detailed in §4.2) and the function used to determine the announcement validity (called '_valid_ann', an example shown in Figure 4).

The ranking of announcements follows the Gao Rexford model by default, but, when needed, can be tailored, for example, to implement policies such as 'security first' or 'security second', see §6.1. The user can also easily modify the tie-breaking mechanisms.

The announcement validity function can be overridden, e.g., to implement policies such as ROV. The default announcement validity function only checks for loop-prevention, i.e., returns a Boolean indicating if the ASN (of the AS class) is contained within the AS Path of the announcement (if so, the announcement is invalid). Notice that default function already allows simulation of path poisoning[35], as used either by attacks or for traffic engineering.

The code in Figure 4 is an example, showing the simple derivation of the ROV-AS validity function from BGP-AS. As shown, this only requires eight lines of code, where we check if the announcement is invalid due to a ROA, and otherwise, simply invoke the default BGP validity function. Other examples can be seen in Figure 9 and Figure 10.

```python
class ROVAS(BGPAS):
    def _valid_ann(self, ann: Ann) -> bool:
        # If ROA is invalid, ROV says announcement is invalid
        if ann.invalid_by_roa:
            return False
        # If ROA is valid, determine validity with BGP
        else:
            return super(ROVAS, self)._valid_ann(ann)
```

Figure 4: A subclass of BGP AS that implements ROV.

•**Scenario Class** The Scenario class, used to control attack and defense strategy, can also be easily extended. Common examples include modifying how attackers and victims are selected, how adopting ASes are selected, how various metrics get recorded, etc. Below we show an example for how easy it is to create a non routed prefix hijack scenario (defined in Table 5). With just 15 lines of code, we can create a completely new attack strategy. Multiple use cases listed in §6 implement a wide variety of custom attacking strategies.

```python
class NonRoutedPrefixHijack(Scenario):
    def _get_announcements(self) -> List["Announcement"]:
        """Returns announcements used to seed the engine"""
        anns = list()
        for attacker_asn in self.attacker_asns:
            anns.append(self.AnnCls(
                prefix='1.2.0.0/16',
                as_path=(attacker_asn,),
                timestamp=1,
                seed_asn=attacker_asn,
                roa_valid_length=True,
                roa_origin=0,
                recv_relationship=Relationships.ORIGIN
            ))
        return anns
```

Figure 5: A scenario subclass that demonstrates how to implement a non routed prefix hijack

•**Announcement Class** The Announcement class (containing the information used for the BGP announcements that are propagated) is often extended to contain attributes specific to each simulation. This can be accomplished easily by simply overriding the init function of the Announcement class in a subclass with just a few lines of code, as shown in Figure 6. This is an example of the announcement class used to simulate the announcements described in ROV++ [43], which contains a few additional attributes, such as the holes in the announcement, whether or not the announcement is a blackhole or a preventive announcement, etc.

```
@dataclass(frozen=True, slots=True)
class ROVPPAnn(Announcement):
    holes: tuple[str] = ()
    blackhole: bool = False
    # V3 attributes
    preventive: bool = False
    attacker_on_route: bool = False
```

Figure 6: A subclass of Announcement that implements additional attributes needed to simulate ROV++ [43]

4 THE BGPY SIMULATION ENGINE

In this section we describe various aspects of the simulation engine, used to simulate BGP with given security policies and attacks. The simulator engine abstracts away packet level interactions, and simulates BGP from a high level, propagating BGP announcements across the AS topology to produce as local RIB at each AS for data analysis within the Simulation Framework (see §3.1).

4.1 AS Graph

To start our simulations, we build an AS Graph. The simulator receives the topology information in the CAIDA serial-2 format [10], and, by default, uses the latest CAIDA topology. This topology is a directed acyclic graph consisting of ASes for nodes and peer to peer connections as well as provider to customer connections for edges.

Nodes. The nodes in this graph are ASes, by default performing BGP (for further detail, see §4.2). We differentiate between four types of ASes:

- **Stubs**. These are ASes that have only one provider and no peers or customers.
- **Multihomed**. These are ASes which do not have any customers, but do have more than one provider, or have one or more peers. Often this is done for backup purposes or load balancing [59].
- **Input Clique**. The CAIDA AS topology [10] provides a strongly connected clique of ASes at the top of the graph with less than 20 ASes.
- **Etc**. These are ASes that don't fit into the other categories

Edges (relationships). We use the standard edges associated with ASes and as defined by the CAIDA topology [10, 24]. Peer-to-peer connections, where traffic flows freely from AS to AS, and customer-provider connections, where providers are paid by their customers to provide traffic. IXPs and sibling relationships are excluded, as these are also excluded in the provided CAIDA topology.

4.2 Vally Free Assumptions and Other Routing Policies

By default the base AS class (BGPAS) uses BGP and adheres to Gao-Rexford valley-free routing rules, which is in line with other works that evaluate the security of inter-domain routing [24]. By default, the BGPAS class also utilizes an export-to-all policy, meaning a route exported to one provider is exported to all providers. The same applies to peers and customers.

Gao-Rexford defines rules for ranking received announcements based on relationship (peer-to-peer or provider-to-customer) and export policies based on those relationships. These rules are usually implemented in routers using the Local Preference mechanism in the BGP Best-Path Selection process, which follows several steps to determine the best path for a particular prefix. The steps are:

• **Local Preference**. The local preference of the AS follows the business incentives of that AS. First, announcements from customers are given the highest priority, because customers are paying for the traffic. Then, announcements from peers are considered, since traffic to and from peers is free. Lastly, announcements from providers are considered. Provider announcements are the lowest for the local preference because this traffic must be paid for.

• **Shortest AS Path**. If multiple announcements have the same relationship, the announcements are ranked by shortest AS Path.

• **Tiebreakers**. If all other aspects of announcements are equal, then the AS defaults to tiebreakers. In our simulations, we default to the lowest ASN to win ties to be consistent with [45] but this can easily be changed and is extendable.

Export Policies. The export policies also follow the business incentives for an AS by default in the BGPAS class. Announcements from customers are sent to peers, providers, and customers. Announcements from both peers and providers are sent only to customers. By default the BGPAS uses an export to all methodology.

Accuracy of the Valley Free Routing model This methodology is one that is widely used across various routing security studies [11, 25, 28], although we do acknowledge that in the real world ASes do not always adhere to this [1, 36, 40, 44]

Prefix Aggregation. The BGPAS class by default does not do any prefix aggregation. That is to say that the BGPAS class will select an announcement for every single prefix, regardless of the existence of subprefixes or superprefixes.

4.3 Real World Data

There are currently several sources of data that provide information on routing policies, particularly for ROV (Route Origin Validation). In Table 4, it is explained that any Scenario class can receive a dictionary of ASN-AS Class key-value pairs as a parameter using the hardcoded_asn_cls_dict. This section provides an explanation of the data sources utilized in a utility function that is integrated within the system. This function generates a hardcoded_asn_cls_dict containing actual ROV data from the real world. This enhancement ensures that real-world ROV can be employed in any simulation, resulting in improved accuracy compared to previous simulators. It is important to highlight that users have the flexibility to input any dataset or routing policy for a given ASN. As additional routing policies become available, they will be incorporated into our datasets and simulations.

• **ROV RPKI**. https://rov.rpki.net/ [50] is a website detailing ASN's and their corresponding ROV confidence.

• **Cloudflare's https://isbgpsafeyet.com**. [14] This website accurately identifies a user's ISP's properties, including its ROV adoption and whether it follows the default policy or a variant that only filters announcements from peers.

• **Revisiting RPKI Route Origin Validation on the Data Plane**. [51] Various metrics were utilized in this study to acquire ROV ASNs along with their corresponding confidence levels.

4.4 Running a Scenario with the Simulation Engine

When we are running a Scenario class , we must first insert the attacker's announcements in the local RIB of the attacker AS, and the victim's announcements in the local RIB of the victim AS. These initial announcements are defined in the Scenario class, for an example see Figure 5. After inserting these announcements, we then run the simulation engine. This performs propagation, to propagate the announcements throughout the internet and the AS topology.

• To start with, we propagate the announcements from customers to providers, all the way up the graph.

• Then, we propagate announcements across peer connections.

• Finally, we propagate announcements from providers to customers.

We perform the propagation of announcements in this way to simulate a moment in time for the AS topology. With this methodology, our simulator converges after a single round of propagation, which significantly reduces the run time of simulations. With this method we avoid a lot of the run time constraints detailed in many other packet-level or discrete event simulators that must converge where simulations required convergence [13, 18, 19, 22, 55], resulting in many rounds of propagation for a single trial, drastically increasing the runtime.

Our method of propagation has time complexity of $O(E)$ per propagation round, where E is the number of edges in the AS topology. By default the AS graph converges after a single round of propagation, however, if there are user-defined routing policies that delay the convergence of the AS graph, the `propagation_rounds` is a parameter that is easily modifiable. This runtime allows us to be able to run our simulations on a standard laptop. For more performance enhancements, future improvements, and various benchmarks, refer to Appendix C.

5 VERIFICATION AND TESTING

Routing attacks and defenses are subtle; it is all too easy to make hard-to-detect errors. Therefore, verification and testing are critical. BGPy includes the following mechanisms to assist in testing and verification.

• **Deterministic randomness** Setting the PYTHON_HASH_SEED in both the Python interpreter's environment and the simulator parameters (which seeds the random module, as shown in Table 3) ensures that running simulations multiple times will yield identical outcomes, despite the initial random generation. This functionality greatly facilitates the reproduction of specific issues and reproducible results that might otherwise be challenging to recreate.

• **YAMLable** Every object in BGPy can be seamlessly converted to and from YAML, providing us with a convenient way to save different parts of the state. For instance, we can easily dump the entire AS graph along with all the announcements into YAML. Additionally, we can save traceback results and other metrics in YAML format. This greatly simplifies the comparison between the YAML generated during testing and the ground truth YAML files we generate ourselves. This functionality is made possible by YAMLable [61], which necessitates custom functions for YAML conversion in each class. We prefer YAML over JSON because it allows direct conversion between Python objects, unlike JSON. While pickling in Python also supports this feature, pickled objects are not easily readable by humans, and pickling highly recursive objects such as the AS graph proved quite challenging. YAML, on the other hand, is both human-readable and supports conversion to and from even the most complex and recursive Python objects.

• **Diagrams** Our test suite includes a feature that allows users to visually represent smaller AS graphs, the local RIB, and various metrics as diagrams prior to running newly implemented routing policies on the full topology. These diagrams are generated using graphviz [2] and serve as valuable tools for debugging. Without a visual representation of the AS topology in small graphs, identifying logic problems becomes extremely challenging.

To generate these diagrams, input requirements include an AS topology and a scenario. The simulation is then executed, resulting in the display of the AS graph and associated metrics.

For instance, consider Figure 7. Each AS in the diagram contains the ASN, policy information (in other words the label of the AS class), and its local RIB. The local RIB table includes the prefix, AS Path, and origin for each announcement. Circular shapes represent BGP ASes (as also indicated in their policies listed), while the octagon shapes denote adoption of a different policy (as seen in Figure 7 with ASes 3 and 4 adopting ROV). ASes 666 (the attacker) and 777 (the victim) have an additional ring around them, signifying that they are the origins of announcements. The arrows between ASes represent provider-to-customer connections, while peer-to-peer connections (shown in Figure 11) are denoted by dashed lines.

The color automatically determined for each AS indicates the outcome of the AS on the data plane for the most specific prefix (in Figure 7, it is /24). Differentiating between the control plane and the data plane is crucial for understanding these outcomes, as they often differ. For example, in the control plane of Figure 7, AS 3 appears unaffected by hijacking, as it does not contain any of the attacker's announcements within it's local RIB. However, upon performing traceback, we discover that on the data plane, traffic for AS 3 is routed to AS 1, which then forwards all traffic within the /24 prefix to AS 666, the attacker. Green represents successfully routed announcements to the victim for the most specific prefix (/24 in Figure 7). Gray (shown in Figure 11) indicates that an AS is disconnected. An aggregated table presents the metrics for attacker success, victim success, and disconnections. Additionally, users have the option to label the graph and provide a description. The emoji on the rightmost column denotes the origin, with the angel being the victim, the 'devil' being the attacker, and a shield being a preventive announcement (as shown in Figure 11)

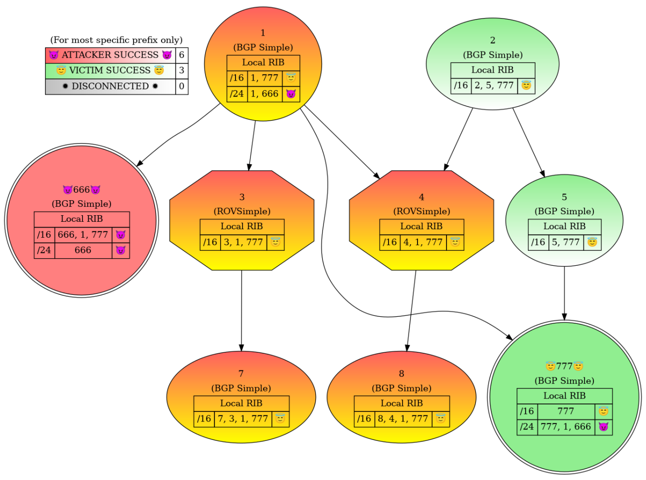

Figure 7: Test Suite Diagram Example that was dynamically generated from an input AS topology and the origin announcements. For a detailed description of this diagram, please refer to §5 under the bullet 'diagrams'. For a larger version, please see Figure 8 in the appendix. For an example of disconnections and peering, see Figure 11.

.

System Test Suite All of the features listed in this section are utilized in our system test suite. A user has the ability to define a custom AS topology, along with a custom scenario, and the system test suite will automatically perform the following:

(1) Creates an AS topology with custom routing policies.
(2) Runs a scenario on said AS topology, such as a subprefix hijack.
(3) Records all the metrics associated with that topology, such as number of ASes hijacked, etc.
(4) Displays this topology and resulting output in a diagram for easy visualization (described in the bullet for diagrams, see an example in Figure 7).
(5) Compares it to ground truth YAML that is saved and vetted in advance.

Additionally, the ground truth would be difficult and time consuming for a user to generate due the vast amount of text that would be required to write these YAML AS graphs by hand. Instead, the ground truth can be auto generated, and the user can simply verify that it is accurate, saving countless hours of manual work.

Across all of our various use cases we have hundreds of system tests. The system tests suite can even be used to verify other simulation engines.

6 USE CASES

This section describes some ways in which BGPy has been used for BGP security research so far. All of these use cases involved extensions to the core BGPy mechanisms and did not require any modification to the simulator source code. Not only do these use cases use the wide variety of parameters available, they also have implemented many of their own AS subclasses for their defensive policies, many different attacker scenarios, subclassed many different aspects of the simulations themselves, etc.

6.1 ROV++ and Mixed Deployment

We simulate the policies described in [43] using BGPy. This only required us to implement a custom announcement class (see Figure 6, an announcement class that contains attributes for holes and preventive announcements) and custom AS classes for ROV++ V1, V2, V3 (along with the corresponding lite versions). The ease of use is important to highlight, as only the ideas described in [43] required implementation, while the rest of BGPy remained untouched without modification to the source code.

For the AS classes themselves, the announcement rank functions were modified to include security policies after local preference but before AS path length and tiebreaks. An example of an ROV++ V2 AS class is included in Figure 10, requiring less than 20 lines of code to implement. Additionally, all of these policies were verified through over 100 new system tests using the existing test suite. An example of one of these tests can be seen in Figure 11.

This work is currently being extended to also include *mixed deployment* scenarios where ROV++ has partial adoption amongst real world ROV ASes. This extension required only a parameter change to pass in a list of real world ROV ASes, without any modification to the simulation code or even the ROV++ code (for a full list of parameters, see Table 4).

6.2 Attacker Collusion

The simulator's versatility is supported by its use in developing and simulating policies that can cope with multi-attacker hijack scenarios; ones of which the attackers may or may not be colluding. This means orchestrating a number of attackers as a group/team to launch an attack on a target to achieve a particular goal, such as decreasing successful connections to the origin.

A policy that deals with such attacks was created by subclassing the `BGP_AS` and override method(s) related to how it processes the announcements. In order to vary the number of attackers, one simply needs to provide an argument of how many attackers are in the system via the `Simulation` class (see Table 3 for a full list of parameters), and the attacker announcements would need to be constructed and coordinated in a subclass of the `Scenario` class.

Auxiliary entities, such as a centralized server, were integral to this policy. Although BGPy doesn't have a centralized server as a component of its architecture, it provides the flexibility to incorporate it. As in this case, the server was created as a class which became a member attribute of the new `Scenario` class.

6.3 Path Security Policies

BGPy has been extended to evaluate defenses against *path manipulation* attacks, where an attacker modifies the AS Path or other attributes of an announcement. Notably, this facilitates evaluation of BGPsec [34] and alternative mechanisms that provide Path Security. The BGPsecAS class extends the default path selection mechanism to validate and prefer signed paths over unsigned paths and extends the default export function to add signatures in outgoing announcements. The Announcement class is extended to support propagating signatures. A new scenario subclass evaluates defenses against *origin hijacks*, where the attacker announces an AS path indicating it is a neighbor of the legitimate origin. These attacks evade detection by ROV since the origin is valid, and may become more common as RPKI/ROV adoption continues to increase. A mechanism like BGPsec is necessary to detect and prevent such attacks.

The Path Security extensions also include support for evaluating defenses against *route leaks* [53], where an AS announces routes in violation of common export policy. Importantly, a route leak may redirect a substantial amount of traffic even without manipulating the AS path. In other words, even if BGPsec was fully deployed, a route leak could still have the potential to cause harm and redirect internet traffic. Separate solutions, such as ASPA [3] are needed to protect against route leaks.

7 CONCLUSION

We presented BGPy, an open-source BGP Python simulator designed to analyze various attack and defense security scenarios https://github.com/jfuruness/bgpy. We showcased the simulator framework and functionality, and demonstrated it's ease of extensibility and use. We detailed the simulation engine and design, documenting how we model the AS topology and it's corresponding routing policies. We also demonstrated the BGPy test suite, and showed how it can make simulations both easy to debug and robust. We cataloged numerous performance metrics and enhancements. We described several use cases where BGPy is already in use, and highlighted it's extendability and usefulness. We will continue to improve upon it's limitations as future work, as described in Appendix A.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions. This manuscript is based upon work supported by the National Science Foundation under Grant No. 2247810. Prof. Herzberg was partially supported by an endowment from Comcast. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or of Comcast.

REFERENCES

[1] Ruwaifa Anwar, Haseeb Niaz, David Choffnes, Italo Cunha, Phillipa Gill, and Ethan Katz-Bassett. 2015. Investigating Interdomain Routing Policies in the Wild. In *Proc. of ACM IMC*.
[2] The NetworkX Developers Aric Hagberg. 2022. Graphviz. https://graphviz.org/
[3] Alexander Azimov, Eugene Bogomazov, Randy Bush, Keyur Patel, and Job Snijders. 2022. *BGP AS_PATH Verification Based on Resource Public Key Infrastructure (RPKI) Autonomous System Provider Authorization (ASPA) Objects*. Internet-Draft draft-ietf-sidrops-aspa-verification-09. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-sidrops-aspa-verification/09/ Work in Progress.
[4] Hitesh Ballani, Paul Francis, and Xinyang Zhang. 2007. A Study of Prefix Hijacking and Interception in the Internet. In *Proc. of ACM SIGCOMM*. 265–276. https://doi.org/10.1145/1282380.1282411
[5] Tony Bates, Geoff Huston, and Philip Smith. [n. d.]. CIDR REPORT for 15 May 23. https://www.cidr-report.org/as2.0/
[6] BGPStream. 2018. BGPMON's BGP Stream incident alert service. https://bgpstream.com.
[7] Jay Borkenhagen. 2019. AT&T/as7018 now drops invalid prefixes from peers. Email. https://mailman.nanog.org/pipermail/nanog/2019-February/099501.html NANOG Mailing List Archive.
[8] Kevin Butler, Toni R. Farley, Patrick McDaniel, and Jennifer Rexford. 2010. A Survey of BGP Security Issues and Solutions. *Proc. IEEE* 98, 1 (2010), 100–122.
[9] CAIDA. [n. d.]. CAIDA BGPStream twitter. https://twitter.com/bgpstream/.
[10] CAIDA. 2016. The CAIDA AS Relationships Dataset. http://www.caida.org/data/as-relationships/.
[11] Haowen Chan, Debabrata Dash, Adrian Perrig, and Hui Zhang. 2006. Modeling Adoptability of Secure BGP Protocols. In *Proc. of SIGCOMM*. ACM.
[12] Zhiguo Chen, Xin Wang, Rui Zhang, Vern Paxson, and Stefan Savage. 2014. Characterizing and detecting malicious behavior in bgp routing. In *Proceedings of the 2014 ACM SIGCOMM conference on computer and communications security*. ACM, 103–114.
[13] Zhe Chen, Daqiang Zhang, and Yinxue Ma. 2015. Modeling and analyzing the convergence property of the BGP routing protocol in SPIN. *Telecommunication Systems* 58 (03 2015). https://doi.org/10.1007/s11235-014-9870-y
[14] Cloudflare. 2023. Is BGP Safe Yet? https://isbgpsafeyet.com/
[15] Avichai Cohen, Yossi Gilad, Amir Herzberg, and Michael Schapira. 2016. Jump-starting BGP security with path-end validation. In *Proc. of ACM SIGCOMM*. ACM, 342–355.
[16] J.H. Cowie, D.M. Nicol, and A.T. Ogielski. 1999. Modeling the global Internet. *Computing in Science & Engineering* 1, 1 (1999).
[17] Remy de Boer and Javy de Koning. 2013. *BGP Origin Validation (RPKI)*. Technical Report. Univeristy of Amsterdam, Systems and Network Engineering Group.
[18] Christos Dimitropoulos and Greg Riley. 2008. SSFNet: A Scalable Simulation Framework for BGP. *IEEE Journal on Selected Areas in Communications* 26, 1 (2008), 158–167. https://doi.org/10.1109/JSAC.2007.357111
[19] Xenofontas A. Dimitropoulos and George F. Riley. 2006. Efficient Large-Scale BGP Simulations. *Comput. Netw.* 50, 12 (aug 2006), 2013–2027. https://doi.org/10.1016/j.comnet.2005.09.033
[20] Nick Feamster, Jonathan Winick, and John Rexford. 2004. A model of BGP routing for network engineering. In *Proceedings of the 2004 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 191–202. https://doi.org/10.1145/1012888.1005726
[21] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, and John Rexford. 2000. Netscope: Traffic engineering for ip networks. *IEEE Network* 14, 2 (2000), 11–19.
[22] Tony Feng and Rob Ballantyne. 2004. Implementation of bgp in a network simulator. *Proc. Applied Telecommunications Symposium, ATS'04* (01 2004).
[23] Python Software Foundation. 2023. *Python 3.11*. https://www.python.org/downloads/release/python-311/
[24] Lixin Gao and Jennifer Rexford. 2001. Stable Internet Routing without Global Coordination. *IEEE/ACM Trans. Netw.* 9, 6 (dec 2001), 681–692. https://doi.org/10.1109/90.974523
[25] Yossi Gilad, Avichai Cohen, Amir Herzberg, Michael Schapira, and Haya Shulman. 2017. Are We There Yet? On RPKI's Deployment and Security. In *NDSS*. The Internet Society.
[26] Phillipa Gill and Nick Feamster. 2012. Modeling on Quicksand: Dealing with the Scarcity of Ground Truth in Interdomain Routing Data. *ACM SIGCOMM Computer Communication Review* 42, 4 (2012), 107–118. https://people.cs.umass.edu/~phillipa/papers/QuickSand.pdf
[27] Phillipa Gill, Michael Schapira, and Sharon Goldberg. 2011. Let the market drive deployment: A strategy for transitioning to BGP security. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 14–25.
[28] Sharon Goldberg, Michael Schapira, Pete Hummon, and Jennifer Rexford. 2014. How secure are secure interdomain routing protocols? *Computer Networks* 70 (2014), 260–287.
[29] Amir Herzberg, Matthias Hollick, and Adrian Perrig. 2015. Secure Routing for Future Communication Networks (Dagstuhl Seminar 15102). *Dagstuhl Reports* 5, 3 (2015), 28–40. https://doi.org/10.4230/DagRep.5.3.28
[30] G. Huston, M. Rossi, and G. Armitage. 2011. Securing BGP: A literature survey. *IEEE Communications Surveys & Tutorials* 13, 2 (2011), 199–222.
[31] Cheng Jin, Qian Chen, and Sugih Jamin. 2000. Inet: Internet Topology Generator. *IEEE/ACM Transactions on Networking* 8 (11 2000), 753–765. Issue 6. https://doi.org/10.1109/TNET.2000.880966
[32] Josh Karlin, Stephanie Forrest, and Jennifer Rexford. 2008. Autonomous security for autonomous systems. *Computer Networks* 52 (10 2008), 2908–2923. https://doi.org/10.1016/j.comnet.2008.06.012
[33] S. Kent and K.seo. 2012. *An Infrastructure to Support Secure Internet Routing*. RFC 6480. The Internet society. http://tools.ietf.org/html/rfc6480
[34] M. Lepinski (Ed.) and K. Sriram (Ed.). 2017. BGPsec Protocol Specification. RFC 8205 (Proposed Standard). https://doi.org/10.17487/RFC8205 Updated by RFC 8206.

[35] Thomas B London, Stephen R Hanna, and Kevin L Carter. 2000. Path poisoning in bgp. In *Proceedings of the 2000 IEEE international conference on network protocols*. IEEE, 191–198.
[36] H. Madhyastha, E. Katz-Bassett, T. Anderson, A. Krishnamurthy, and A. Venkataramani. 2009. iPlane Nano: Path Prediction for Peer-to-Peer Applications. In *Proce of NSDI*.
[37] Mohammadreza Maleki, Mohammad Mahdi Hajian, and Mohammad R Sadeghi. 2018. BGP anomalies: A survey of detection methods and their limitations. *IEEE Communications Surveys & Tutorials* 20, 3 (2018), 2028–2055.
[38] Niko Matsakis. 2023. PyO3: Bringing Python to Rust. https://github.com/PyO3/pyo3.
[39] Jared Mauch. [n. d.]. BGP Routing Leak Detection System. https://puck.nether.net/bgp/leakinfo.cgi/leaks-majornet.txt.
[40] R. Mazloum, M. Buob, J. Auge, B. Baynat, D. Rossi, and T. Friedman. 2014. Violation of Interdomain Routing Assumptions. In *Proc. of Passive and Active Measurement Conference (PAM)*.
[41] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. 2001. BRITE: an approach to universal topology generation. In *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 346–353. https://doi.org/10.1109/MASCOT.2001.948886
[42] Asya Mitseva, Andriy Panchenko, and Thomas Engel. 2018. The state of affairs in BGP security: A survey of attacks and defenses. *Computer Communications* 124 (June 2018), 45–60.
[43] Reynaldo Morillo, Justin Furuness, Amir Herzberg, Cameron Morris, James Breslin, and Bing Wang. 2020. ROV++: Improved Deployable Defense against BGP Hijacking. In *Proceedings of the Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2021.24438
[44] W. Mühlbauer, A. Feldmann, O. Maennel, M. Roughan, and S. Uhlig. 2006. Building an AS-topology model that captures route diversity. In *Proc. of SIGCOMM*.
[45] C. Perkins, P. Calhoun, and J. Bharatia. 2007. Mobile IPv4 Challenge/Response Extensions (Revised). RFC 4721 (Proposed Standard). https://doi.org/10.17487/RFC4721
[46] Brian J. Premore. 2003. *An Analysis of Convergence Properties of the Border Gateway Protocol Using Discrete Event Simulation*. Ph. D. Dissertation. Dartmouth College.
[47] B. Quoitin and S. Uhlig. 2005. Modeling the routing of an autonomous system with C-BGP. *IEEE Network* 19, 6 (2005), 12–19. https://doi.org/10.1109/MNET.2005.1541716
[48] Y. Rekhter (Ed.), T. Li (Ed.), and S. Hares (Ed.). 2006. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard). https://doi.org/10.17487/RFC4271 Updated by RFCs 6286, 6608, 6793, 7606, 7607, 7705, 8212, 8654, 9072.
[49] Renesys. [n. d.]. The New Threat: Targeted Internet Traffic Misdirection. http://www.renesys.com/2013/11/mitm-internet-hijacking/.
[50] Andreas Reuter, Randy Bush, Italo Cunha, Ethan Katz-Bassett, Thomas C Schmidt, and Matthias Wählisch. 2018. Towards a rigorous methodology for measuring adoption of RPKI route validation and filtering. *ACM SIGCOMM Computer Communication Review* 48, 1 (2018), 19–27. Online service: https://rov.rpki.net/.
[51] Nils Rodday, Ítalo S. Cunha, Randy Bush, Ethan Katz-Bassett, Gabi Dreo Rodosek, Thomas C. Schmidt, and Matthias Wählisch. 2021. Revisiting RPKI Route Origin Validation on the Data Plane. In *5th Network Traffic Measurement and Analysis Conference, TMA 2021, Virtual Event, September 14-15, 2021*, Vaibhav Bajpai, Hamed Haddadi, and Oliver Hohlfeld (Eds.). IFIP. http://dl.ifip.org/db/conf/tma/tma2021/tma2021-paper11.pdf
[52] Muhammad S. Siddiqui, Diego Montero, Rene Serral-Gracia, Xavi Masip-Bruin, and Marcelo Yannuzzi. 2015. A survey on the recent efforts of the Internet Standardization Body for securing inter-domain routing. *Computer Networks* 80 (April 2015), 1–26.
[53] K. Sriram, D. Montgomery, D. McPherson, E. Osterweil, and B. Dickson. 2016. Problem Definition and Classification of BGP Route Leaks. RFC 7908 (Informational). https://doi.org/10.17487/RFC7908
[54] Cisco Systems. 2023. *Cisco WAN Automation Engine (WAE)*. https://www.cisco.com/c/en/us/products/routers/wan-automation-engine/index.html
[55] Boleslaw Szymanski, Yu Liu, and Rashim Gupta. 2003. Parallel network simulation under distributed Genesis. In *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 61–68. https://doi.org/10.1145/824475.825869
[56] PyPy Team. 2023. *PyPy*. https://pypy.org/
[57] Andree Toonk. 2011. Indosat a Quick Report. http://www.bgpmon.net/hijack-by-as4761-indosat-a-quick-report/.
[58] Andree Toonk. 2014. Turkey Hijacking IP Addresses for Popular Global DNS Providers. BGPMon.
[59] Maarten C van Steen, Arno P Akkermans, and Henk J Sips. 2008. Multihoming in the internet: A survey. *IEEE Communications Surveys & Tutorials* 10, 2 (2008), 372–392.
[60] Pierre-Antoine Vervier, Olivier Thonnard, and Marc Dacier. 2015. Mind Your Blocks: On the Stealthiness of Malicious BGP Hijacks. In *NDSS*.
[61] Jacob Vlijm. 2022. yamlable. https://github.com/jacobvlijm/yamlable
[62] Matthias Wählisch, Olaf Maennel, and Thomas C. Schmidt. 2012. Towards detecting BGP route hijacking using the RPKI. In *Proc. of ACM SIGCOMM*. 103–104. https://doi.org/10.1145/2342356.2342381
[63] M. Wojciechowski. 2008. *Border gateway protocol modeling and simulation*. Master's thesis. University of Warsaw.
[64] J. Wu, Y. Zhang, Z. M. Mao, and K. Shin. 2007. Internet routing resilience to failures: Analysis and implications. In *Proc. of CoNEXT*.
[65] Ellen W Zegura, Kenneth L Calvert, and Samrat Bhattacharjee. 1996. How to model an internetwork. In *Proceedings of the IEEE conference on computer communications (INFOCOM)*, Vol. 2. IEEE, 594–602.
[66] Rui Zhang, Zhiguo Chen, Xin Wang, Vern Paxson, and Stefan Savage. 2015. BGPMon: A system for monitoring and detecting malicious behavior in bgp. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1591–1602.

A FUTURE WORK AND LIMITATIONS

BGPy is also currently being iterated on, and we leave a few improvements as future work.

• **Python** BGPy is implemented in Python [23], which unlike lower level languages has higher runtime and memory constraints. We recommend for BGPy to set aside 1-2GB per core used in simulations and for 1000 trials it takes about 16 minutes per policy per core. In our experience working on these types of simulations, the pros with the faster development time and flexible extensions far outweighs the negatives of a higher level language. We found that developer time was the main constraint as we wanted to evaluate security ideas quickly. However, we plan on improving on this, and we believe that we can use Rust bindings using the PyO3 crate [38] for the speed of a lower level language while still maintaining the flexibility and ease of development of a higher level language. See a more in depth discussion in C.

• **Discrete Event Simulations** BGPy is not designed to be a discrete event simulator. BGPy assumes a completely synchronous system for each propagation round for efficiency purposes. By avoiding discrete event simulations, we can avoid the need to allow the AS graph to converge, which significantly reduces the runtime for these simulations. However, as a parameter in the simulator, a user of the simulator can set the amount of rounds of propagation, so in theory they can do as many as they need. BGPy is also extendable, and can be extended to propagate until convergence, and we can add this as a core feature if other developers wish. That being said, propagating more rounds quickly becomes infeasible due to runtime constraints. We hope to revisit this as future work once the simulator contains Rust bindings using the PyO3 crate [38]. See C for a more in depth discussion on performance enhancements.

• **High Level BGP** BGPy focuses on high level aspects of BGP, such as best announcement selection, import and export policy, etc. Due to runtime constraints we do not simulate many of the lower level BGP aspects. We also have limited knowledge of the AS graph. For example, the CAIDA topology [10] does not include sibling relationships or IXPs. We do not include information about prefix aggregation, AS blacklists, etc. However, all of these can be addressed and added by a user to any level of granular detail. The graph and routing policies are extendable, and just like we have added support for ROV, with the appropriate data sources a user can add support for any number of these enhancements. We leave this as future work.

B TEST SUITE EXAMPLE

In this section of the appendix, we showcase an example of one of the diagrams auto generated from a single system test in figure 8, a larger version of Figure 7. There are hundreds like it across the many use cases the simulator is used for. For a detailed description of this diagram, please refer to 5 under the bullet 'diagrams'

C PERFORMANCE AND BENCHMARKS

The Internet is large, with over 80,000 ASes [5], hence, simulations of BGP attacks and defenses can require significant computation time. This can be a concern, especially considering that BGPy uses Python, an interpreted language which is not as efficient as compiled language such as C or C++. However, we found that BGPy is sufficiently efficient, to allow simulations on standard laptops to complete in reasonable time frames. For our benchmarks (in Table 1), we used the following parameters defined in Table 2:

Python 3.11 [23] was used as the default interpreter, for a total runtime of about 30 minutes per adopted policy on a laptop. We highly recommend the use of PyPy [56], a just-in-time Python compiler that requires no changes to the Python code. With PyPy, on the same machine, the total runtime was about 16 minutes per adopted policy.

We also showcase how the simulations scale linearly with the number of CPU cores, which is made possible because trials are treated as independent so each CPU core can run any subset of trials. We present several instances where utilizing cloud compute services can drastically speed up runtimes. We recommend allowing for 1-2GB per core for RAM.

Optimizations. The performance breakdown reveals that 70% of the total run time is dedicated to running the simulation engine and propagating the announcements. Another 25% is allocated to performing the traceback and data analysis (refer to §3.1). Interestingly, BGPy employs a naive recursive traceback approach (without memoization) that may trace back the same sub-path multiple times. Surprisingly, we discovered that this method is more efficient than storing the results for already traced paths. The reason behind this lies in the shallow nature of the AS graph, where the average path length is just 4. Thus, the savings achieved from reduced tracebacks outweigh the overhead of storage and lookup operations, including hashing.

In the real world, it is common for each autonomous system (AS) to calculate the ROV validity for every announcement, resulting in a run time of O(V*A), where V represents the number of ASes (vertices) and A denotes the number of announcements. To avoid this lengthy computation, for our simulations we simplify the process by determining the ROV validity at the originating AS and adding it as an attribute to the announcement. As the announcement spreads across the internet, the ROV validity remains constant since it is based on the prefix-origin pair. Consequently, there is no need to recalculate the ROV validity, reducing the run time for this calculation to O(A). Typically, our simulations involve only one or two announcements at the victim and attacker, making this operation exceptionally fast.

Anticipated Performance Enhancements .Our performance profiling has revealed that more than 70% of the overall run time is dedicated to executing the simulation engine and duplicating announcements across the graph. However, due to the inherent slowness of constructing these objects in Python, we have devised a plan for performance enhancements for future work. Our strategy involves transitioning to Rust bindings within Python using powerful tools like PyO3 [38]. By leveraging this approach, we will be able to facilitate a Python package that incorporates Rust's capabilities. This transition is expected to yield significant performance enhancements without sacrificing extensibility whilst maintaining all of the functionality of the current implementation.

Tradeoffs between speed and ease of use. The BGP Routing Information Base (RIB) contains the routes known by a BGP router, and it is divided into three parts. The Adj-RIBs-In holds all routes received from neighbors, the Local RIB holds the currently selected best routes, and the Adj-RIBs-Out holds routes selected for advertisement to neighbors. The BGP RFC [48] emphasizes that the distinction between these parts is purely conceptual and that implementations need not actually maintain separate copies of the data in memory. Omitting copies of data can substantially decrease run time and memory use, however, in some cases it is convenient to have these data structures easily accessible. Searching for alternate routes, for example, is straightforward when the Adj-RIBs-In is available. Similarly, withdrawn routes can be easily computed from the Adj-RIBs-Out.

For this reason, we offer users a choice between two base BGP ASes. The `BGPAS` class maintains copies of the RIB data structures as they are defined in the RFC. This class is best suited for policies that require multiple rounds of propagation and may cause withdrawn routes. We also offer a light-weight alternative class, `BGPSimple`. The `BGPSimple` class only has a Local RIB and does not simulate withdrawn routes, the Adj-RIBs-In, or the Adj-RIBs-Out. For many defenses, including ROV and the policies defined in [43], the simplified AS model is sufficient. This drastically reduces the complexity of simulating scenarios with these policies, saving time and memory by avoiding the operations to populate those data structures.

Other simulators in the past have employed various techniques for speedups, such as modifying the AS graph to reduce it in size [31, 41, 65]. This can be done by doing things such as removing stub ASes (since they should have identical Local RIBs to their providers), amongst other techniques. For our simulations we do not employ these techniques, since certain policies and scenarios may utilize the ASes that were removed for attack or defense, based on their respective policy.

D SIMULATOR AND SCENARIO PARAMETERS

In this section of the appendix, we include parameters to the simulator (Table 3) and to the scenario class (Table 4). We also showcase a table of provided scenario's in Table 5.

In the simulator parameters, we detail various options that affect the entire simulation and all scenarios contained within that simulation. For instance, by setting the number of trials to 1000, all scenarios will be run 1000 times.

In contrast, each scenario that is being compared is also highly configurable. As in the example showcased in Figure 3, one scenario has an adopting AS of ROV, and a different scenario has an adopting AS of a different ROV variant that filters only by peers.

Table 1: Benchmark tests. See Table 2 for benchmark parameters

Hardware	CPUs	RAM	Peak RAM	Interpreter	Runtime per policy
ThinkPad P1 Gen 3, CPU model Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz	12	32GB	6GB	Python 3.11	59 minutes
ThinkPad P1 Gen 3, CPU model Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz	12	32GB	10GB	PyPy 3.10	16 minutes
c6a.32xlarge AWS EC2 instance, CPU model AMD EPYC 7R13 Processor	128	256GB	83GB	PyPy 3.10	< 1 minute
High Performance Computing Cluster with AMD EPYC 128	66	298GB	Unknown	Python 3.11	15 minutes

Table 2: Parameters for benchmarks

Attack Strategy:	Subprefix Hijack
Adopted AS Class:	ROV and an ROV variant that filters only peers
Percent Adoptions:	ONLY_ONE_AS AS, 10%, 20%, 40%, 80%, ALL_BUT_ONE_AS
Trials:	1000
Assertions:	Off (-O flag)

Parameter	Description
`percent_adoptions`	A list of percentages representing the adoption rate of the defensive policy among ASes (Autonomous Systems). Special options are available to set the adoption rate to 0% (with only one AS adopting) or 100% (with all but one AS adopting).
`scenario`	A list of configuration to be analyzed in the simulation. For example, analyzing the effects of subprefix hijack by an attacker and the adoption of ROV (Route Origin Validation) by the victim. Refer to table 5 for the default available scenario options, and see 3 for usage examples
`num_trials`	The total number of trials to be executed. Even a small number of trials (e.g., 100) can provide a clear understanding of the graph, but we typically recommend running 1000 or more trials in our simulations.
`propagation_rounds`	The number of rounds of propagation. In most cases, the graph converges after a single round of propagation. However, in certain scenarios, multiple rounds are necessary. An attacker that behaves differently based on how other ASes respond to its attack, for example, would require multiple rounds to simulate.
`output_path`	Specifies the output path for the generated graphs.
`parse_cpus`	The number of CPU cores to be utilized for multiprocessing. For the scenario's included by default, approximately 1-2GB of memory is required per core.
`python_hash_seed`	When set to an integer, enables deterministic runs. The AS graph is complex, and certain edge cases may only arise when running thousands of trials. This option facilitates debugging such problems and enables reproducibility.

Table 3: Parameters for the Simulator. To see an example of usage, see figure 3. Simulator parameters affect the entire simulation and all scenarios.

The wide variety of parameters offer ease of customization to each unique simulation without requiring any code changes.

E PEER ROV

In this section of the appendix, we describe PeerROV. PeerROV is an ROV variant that only drops announcements invalid by ROA that are received from peers. Several ISPs deploy this variant of ROV, notably, AT&T [7], Zayo, and Digital Energy Technologies Limited (Global) [14]. The code for PeerROV can be seen in Figure 9. When simulated, we found that this ROV variant was insecure and ineffective. This was mainly due to hijacks often being received from customers and providers, so this routing security policy offered little benefit, as shown in Figure 2.

F ROV++

In this section of the appendix, we detail further extensions that were made to support the ROV++ policies described in [43] that did not fit in the main text. The announcement class was extended to support the blackhole and preventive attributes required for ROV++ V1, V2, and V3, and can be seen in Figure 6. The ROVAS class was also extended to support V1, V2, V3, and the corresponding lite versions that were described in [43]. An example of ROV++ V2 can be seen in Figure 10. The system test suite was also used to

Parameter	Description
`AnnCls`	Announcement class to be used in the simulation. This allows users to easily create their own announcements with additional path attributes. This ability have been used in several use cases including the one described in section § 6.1).
`BaseASCls`	Base AS class to be used in the simulation. This is the default class that all ASes will adopt unless otherwise specified in the `hardcoded_asn_cls_dict`. The default base AS class is BGP.
`AdoptASCls`	This is the adopting AS class that will be adopted at each data point for the specified `percent_adoptions` in the simulation parameters. For example, at 5% adoption, 5% of the AS graph will adopt this class for its routing policies. One example of this would be ROV (Route Origin Validation).
`num_attackers`	The number of attackers that will be randomly selected. The default value is one attacker.
`num_victims`	The number of victims that will be randomly selected to announce the legitimate announcement. The default value is one victim.
`hardcoded_asn_cls_dict`	A dictionary of key-value pairs, where the key is the ASN (Autonomous System Number), and the value is the class that the ASN should adopt. By default, this dictionary is empty. For example, there is a utility function included that can pass in the real-world ROV ASes to be set into this dictionary.
`adopting_subcategories`	This is a collection of subcategories in which the adoption will be evenly distributed. Presently, it comprises stubs and multihomed ASes, input clique ASes, and the remaining ASes referred to as "etc ASes." To clarify, if we specify a 10% adoption rate for ASes, it means that 10% of stubs and multihomed ASes, 10% of input clique ASes, and 10% of etc ASes will adopt.

Table 4: Additional Parameters for the scenarios contained within the simulation. To see an example of usage, see Figure 3. Scenario parameters affect only a single scenario that is being compared (for example, one scenario may adopt ROV, while the other scenario may adopt a different ROV variant), not the entire simulation.

Scenario	Description
`PrefixHijack`	Both the attacker and the victim announce the same prefix. The victim's announcement is covered by a ROA.
`SubprefixHijack`	The victim announces a prefix that is covered by a ROA. The attacker announces a subprefix of this.
`NonRoutedPrefixHijack`	The attacker announces a non-routed prefix that is not covered by a ROA (Route Origin Authorization), while the victim announces nothing.
`NonRoutedSuperprefixHijack`	The attacker announces the superprefix of a non-routed prefix. The non-routed prefix is covered by a ROA; however, the superprefix is not. The victim announces nothing.
`NonRoutedSuperprefixPrefixHijack`	The attacker announces the superprefix of a non-routed prefix, as well as the non-routed prefix. The non-routed prefix is covered by a ROA; however, the superprefix is not. The victim announces nothing.
`SuperprefixPrefixHijack`	The victim announces a prefix that is covered by a ROA. The attacker announces the same prefix as the victim, as well as a superprefix that is not covered by a ROA.

Table 5: Default scenarios included in the simulator. Hijacks come from [43, 60].

create over 100 examples of ROV++ policies being used in various scenarios, and an example is included in Figure 11

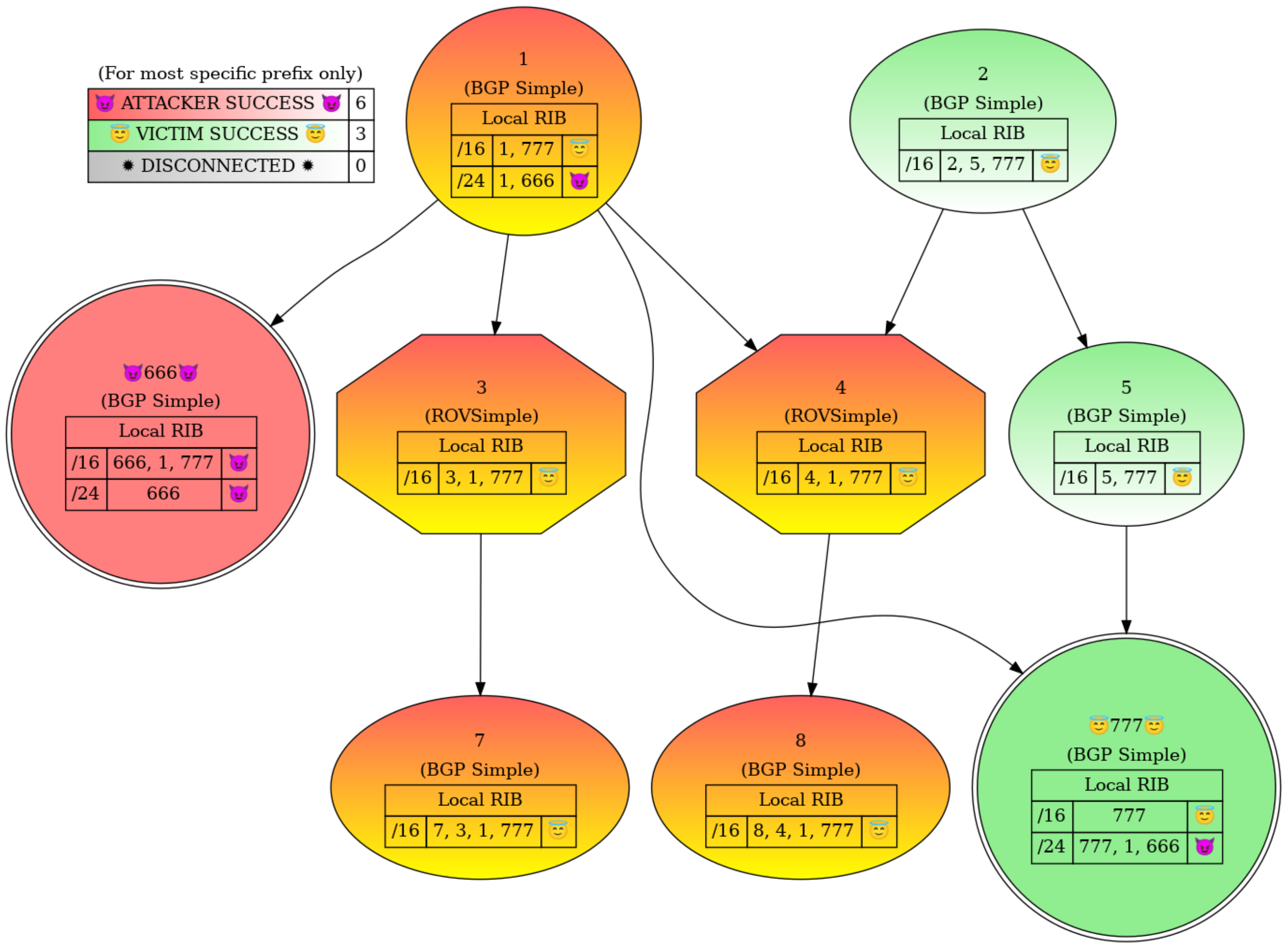
(For most specific prefix only)
😈 ATTACKER SUCCESS 😈 6
😇 VICTIM SUCCESS 😇 3
✹ DISCONNECTED ✹ 0
1 (BGP Simple) Local RIB /16 1, 777 😇 /24 1, 666 😈
2 (BGP Simple) Local RIB /16 2, 5, 777 😇
😈666😈 (BGP Simple) Local RIB /16 666, 1, 777 😈 /24 666 😈
3 (ROVSimple) Local RIB /16 3, 1, 777 😇
4 (ROVSimple) Local RIB /16 4, 1, 777 😇
5 (BGP Simple) Local RIB /16 5, 777 😇
7 (BGP Simple) Local RIB /16 7, 3, 1, 777 😇
8 (BGP Simple) Local RIB /16 8, 4, 1, 777 😇
😇777😇 (BGP Simple) Local RIB /16 777 😇 /24 777, 1, 666 😈

Figure 8: Test Suite Diagram Example. This diagram was dynamically generated from an input of an AS topology and the two originating announcements (at AS 666 and AS 777). For a detailed description of this diagram, please refer to §5 under the bullet 'diagrams'. For a more complex example containing disconnections, peers, and different types of announcements, see Figure 11.

```
class PeerROVAS(BGPAS):
    def _valid_ann(self, ann: Ann) -> bool:
        # If ROA is invalid and announcement is from a peer this ROV variant says the announcement is invalid
        if (ann.invalid_by_roa and ann.recv_relationship == Relationships.PEERS):
            return False
        # If ROA is valid or announcement is not from a peer, determine validity with BGP
        else:
            return super(PeerROVAS, self)._valid_ann(ann)
```

Figure 9: A subclass of BGPAS that implements PeerROV, an ROV variant that only filters announcements received from peers.

```python
class ROVPPV2LiteSimpleAS(ROVPPV1LiteSimpleAS):

    def _policy_propagate(self, neighbor, ann, propagate_to, *args):
        """Deals with blackhole propagation"""

        if ann.blackhole:
            if self._send_competing_hijack_allowed(ann, propagate_to):
                self._process_outgoing_ann(neighbor, ann, propagate_to, *args)
            return True
        else:
            return False

    def _send_competing_hijack_allowed(self, ann, propagate_to):
        return (ann.recv_relationship in [Relationships.PEERS,
                                          Relationships.PROVIDERS,
                                          Relationships.ORIGIN]
                and propagate_to == Relationships.CUSTOMERS
                and (not ann.roa_valid_length or not ann.roa_routed))
```

Figure 10: A subclass that implements ROV++ V2 [43].

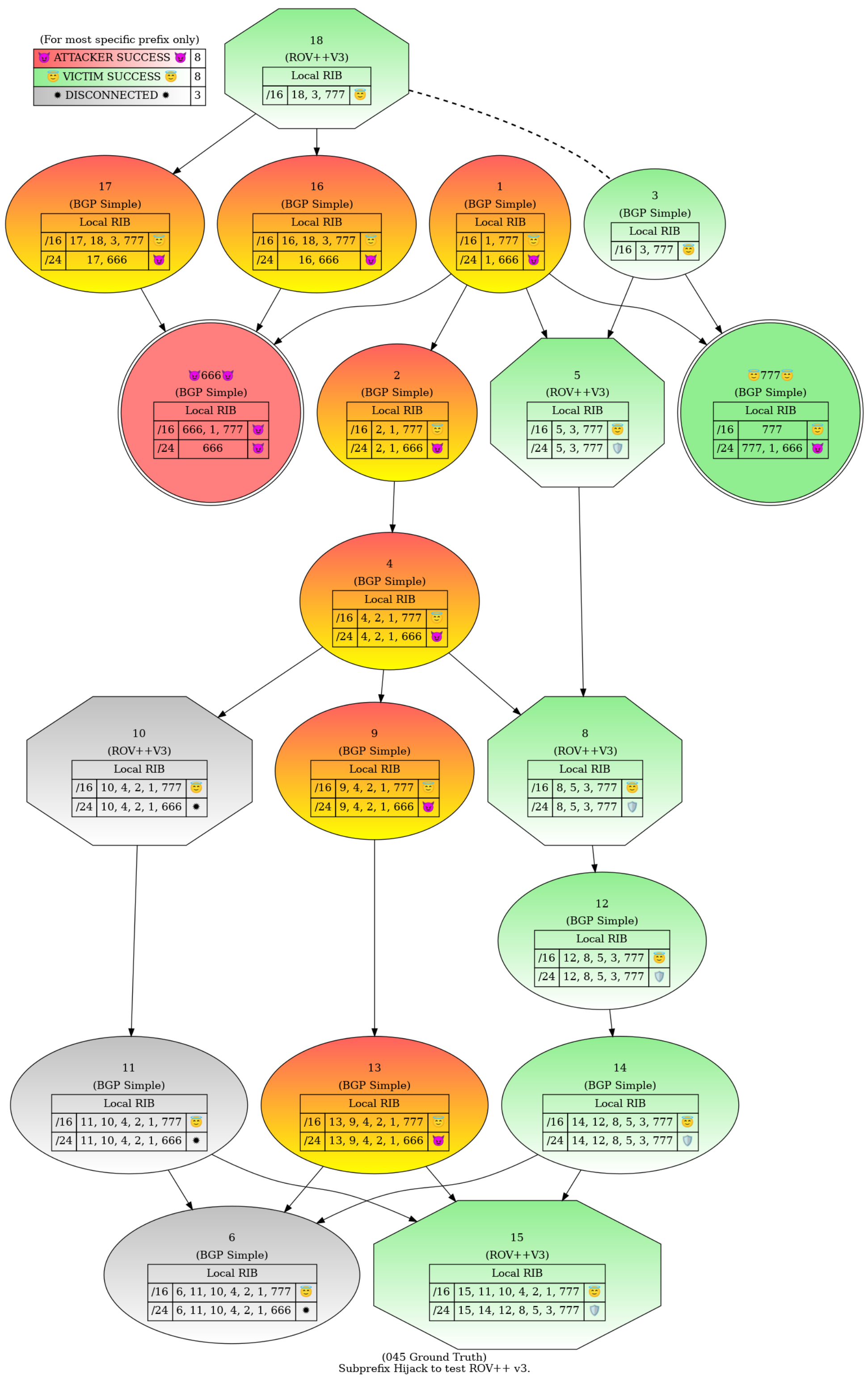
(For most specific prefix only)
😈 ATTACKER SUCCESS 😈 8
😇 VICTIM SUCCESS 😇 8
● DISCONNECTED ● 3
18 (ROV++V3) Local RIB /16 18, 3, 777 😇
17 (BGP Simple) Local RIB /16 17, 18, 3, 777 😇 /24 17, 666 😈
16 (BGP Simple) Local RIB /16 16, 18, 3, 777 😇 /24 16, 666 😈
1 (BGP Simple) Local RIB /16 1, 777 😇 /24 1, 666 😈
3 (BGP Simple) Local RIB /16 3, 777 😇
😈666😈 (BGP Simple) Local RIB /16 666, 1, 777 😈 /24 666 😈
2 (BGP Simple) Local RIB /16 2, 1, 777 😇 /24 2, 1, 666 😈
5 (ROV++V3) Local RIB /16 5, 3, 777 😇 /24 5, 3, 777 🛡️
😇777😇 (BGP Simple) Local RIB /16 777 😇 /24 777, 1, 666 😈
4 (BGP Simple) Local RIB /16 4, 2, 1, 777 😇 /24 4, 2, 1, 666 😈
10 (ROV++V3) Local RIB /16 10, 4, 2, 1, 777 😇 /24 10, 4, 2, 1, 666 ●
9 (BGP Simple) Local RIB /16 9, 4, 2, 1, 777 😇 /24 9, 4, 2, 1, 666 😈
8 (ROV++V3) Local RIB /16 8, 5, 3, 777 😇 /24 8, 5, 3, 777 🛡️
12 (BGP Simple) Local RIB /16 12, 8, 5, 3, 777 😇 /24 12, 8, 5, 3, 777 🛡️
11 (BGP Simple) Local RIB /16 11, 10, 4, 2, 1, 777 😇 /24 11, 10, 4, 2, 1, 666 ●
13 (BGP Simple) Local RIB /16 13, 9, 4, 2, 1, 777 😇 /24 13, 9, 4, 2, 1, 666 😈
14 (BGP Simple) Local RIB /16 14, 12, 8, 5, 3, 777 😇 /24 14, 12, 8, 5, 3, 777 🛡️
6 (BGP Simple) Local RIB /16 6, 11, 10, 4, 2, 1, 777 😇 /24 6, 11, 10, 4, 2, 1, 666 ●
15 (ROV++V3) Local RIB /16 15, 11, 10, 4, 2, 1, 777 😇 /24 15, 14, 12, 8, 5, 3, 777 🛡️
(045 Ground Truth)
Subprefix Hijack to test ROV++ v3.

Figure 11: ROV++ Test Suite Diagram Example. This diagram was dynamically generated as a test case to ensure the correct functionality of ROV++ V3 described in [43]. The gray indicates disconnected. The shield indicates preventive announcements from ROV++ V3. The blackhole indicates a blackhole announcement.