Formal Firewall Discard Verification

MATTHEW JANSEN, *Oregon State University*, Corvallis, Or.

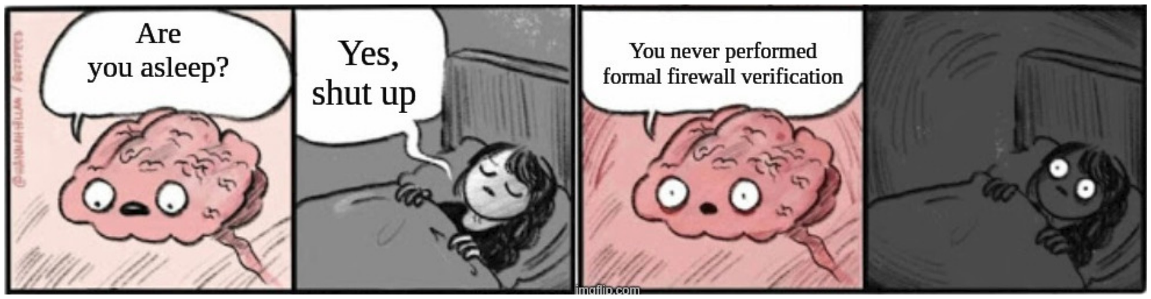

Fig. 1. Accurate depiction of a network security engineer forgetting to verify their firewall rules

Firewalls are used widely to ensure only authorized network traffic is allowed into, or out of a computer network. For a default-deny firewall policy, all network traffic that is not matched by a rule in the policy is dropped. Verifying that default-deny firewall policies are designed according to specification can be trivial for small rulesets, however this problem grows exponentially in time complexity as the number of rules and the number of fields checked increases. Previous work has shown that the problem of verifying that a firewall rule denies a subset of packets that are denied by another firewall (Firewall discard verification, or FV-D) is NP-Hard. In this paper, we explore the FV-D problem, and show that instances of this problem are polynomial-time reducable to UNSAT. Further, we show how this problem can be solved using a SMT solver by introducing *verifire*[1], a tool which solves instances of FV-D by reducing CSV firewalls to boolean formulas and checking if the formula is unsatisfiable.

Keywords: firewalls, network security, formal verification, UNSAT, SMT Solver

1 INTRODUCTION

Information security incidents have caused havoc to corporations and governments for many years, and with the growing online presence of organizations outpacing their information security budgets, the global cost of data breaches has been rapidly increasing. It's been estimated that the global cost of data breaches will exceed $6 trillion annually by the end of 2021, which is a 100% increase from $3 trillion back in 2015 [11]. The impact of information security incidents on the geopolitical landscape is also evident, as nations are funding hacking groups to perform attacks on other nations, one case of this happening in 2021 is the supply-chain attack performed on American corporation SolarWinds, which was believed to have been linked to Russian state-sponsored cyber actors. This resulted in sanctions being placed against Russia, and President Biden signing executive orders to strengthen U.S. cybersecurity [10]. The importance of securing infrastructure against cyber actors is as high as ever, and although the information security budget of many

[1] https://github.com/jansemat/verifire

organizations does not reflect this, deploying infrastructure such as network intrusion detection systems, firewalls, etc. must be of high priority.

Of the pieces of infrastructure that can assist in securing a computer network environment, a network-based firewall is of the most intuitive and important. A firewall's job is to monitor traffic arriving/leaving a network, and to allow or reject the traffic depending on the ruleset implemented within the firewall. These rulesets, or "policies," can be lenient and allow most traffic in/out of a network, or be strict and only allow a very specific set of traffic in/out of the network. The strictness of a policy will depend on the importance of the data being held or transferred by hosts on the computer network - for example, a computer network of hosts that are processing and/or transferring banking information will need to be more "locked down," compared to a computer network whose hosts do not hold sensitive or business critical information.

In order to prove that a firewall will work according to the specifications created and implemented by a network security engineer, it must be formally verified. This is important to organizations because having legitimate network traffic being dropped, or having malicious network traffic being allowed could negatively impact business operations, or increase the organizations attack surface. Formal verification can be done using network traffic simulation tools, however the time complexity of this operation increases exponentially as the number of rules per policy and number of fields per rule increases. In this paper, we show that performing discard verification between two firewalls can be reduced to the boolean unsatisfiability problem (UNSAT). Additionally, we present *verifire*, a firewall policy verification tool which utilizes a SMT solver to convert default-deny firewall verification problem instances into boolean formulas, and attempt to resolve them using the built-in solving methods. Finally we examine the results of this solver as the rules per policy increases.

2 BACKGROUND

This section will discuss related previous works, as well as the terms needed to thoroughly understand the inner-workings of a firewall.

2.1 Related Work

In [4], 13 problems related to the verification, implication, equivalence, adequacy, redundancy and completeness of firewalls were introduced and shown to be NP-Hard by transitive reduction from the boolean 3-satisfiability problem (3-SAT). One of those problems is the firewall discard-verification problem (FV-D), which is the central problem of this paper. In [7], many of these problems (including FV-D) were shown to be resolved by synthesizing firewall policies into an automoton. In [12], firewall verification, synthesis and redundancy problems are solved using a SAT based technique that can compare the equivalence and inclusion relationship between two firewalls. In [5], the use of SAT solvers to solve instances of firewall reachability and cyclicity are investigated and compared to binary decision diagrams (BDDs) for checking the satisfiability of quantified boolean formulae. Additionally, the researchers describe how a general firewall policy may be converted into a boolean formula. In [2], the task of firewall verification and firewall redundancy checking are shown to be equivalent tasks by creating solutions for one problem using another problem's algorithm. Finally in [1], probabilistic algorithms which reduce the runtime of firewall verification tasks to linear time are explored, which entail a reduced time complexity when compared to their deterministic counterparts.

2.2 Definitions and Nomenclature

Many of the definitions discussed here are motivated from [4] and [12]. When a firewall receives a TCP packet from the network, the firewall will begin reading the packet's header information, and will decide to allow or deny the packet to the network after comparing the header's values to its ruleset. The data stored in a packet header will contain data such as source and destination IP addresses, port numbers, protocol numbers, etc. Moving forward, we will consider a TCP packet that is processed by a firewall as a series of values, where each value corresponds to a particular descriptor (e.g. the source IP address), and each value can either be represented by an IPv4 address, or an integer. Specifically, the values which may be represented as an integer are in the range $[0, 2^i - 1]$, where i is a natural number which is dependent on the value's descriptor. For example, the "protocol" value in a TCP packet may be represented by an integer in the range $[0, 2^8 - 1]$. Formally, we define a TCP packet as $pkt = \{(d_1 : v_1), ..., (d_k : v_k)\}$, where each $(d_i : v_i)$ corresponds to a particular descriptor d_i, and the value read from the network packet v_i.

A firewall rule contains two parts, a predicate and a decision. The predicate is a series of descriptors and IPv4 or integer ranges, such that there is exactly one IPv4/integer range for each descriptor in a rule. The decision of a firewall rule is either 1 or 0, which represents a packet either being allowed or denied onto the network, respectively. For a given network packet pkt, for every $(d_i : v_i) \in pkt$, if v_i exists in the range of values given by a firewall rule for descriptor d_i, then the decision corresponding to that rule will be enacted by the firewall onto the network packet in question. A deny-rule is a firewall rule which discards matching packets, and an allow-rule is a firewall rule which allows matching packets onto the network. Formally, we define a firewall rule as $r = \langle \textit{predicate, decision} \rangle$, where

$$\textit{predicate} = \{(d_1 : [a_1, b_1]), \ldots, (d_k : [a_k, b_k])\}; \quad \textit{decision} \in \{0, 1\}$$

A network packet pkt will be matched by rule r if for each $(d_i : v_i) \in pkt$, $[v_1 \in [a_1, b_1] \wedge \cdots \wedge v_k \in [a_k, b_k]] = \textit{True}$, where k is the number of field that are being checked by the firewall. In other words, a packet must match each field in a rule in order for the packet to match the rule. If a rule is matched by a packet, then the packet will be denied access to the network if $\textit{decision} = 0$, and allowed into the network if $\textit{decision} = 1$.

The definition of a firewall, a policy, and a firewall policy are synonymous in this paper, such that they all can be defined by a list of ordered firewall rules. The order of rules within a policy matters, because a packet will be decided on based on the first rule the packet matches within the policy. A discard slice is a firewall policy which contains ≥ 0 allow-rules, followed by a final deny-rule. Similarly, an accept slice contains ≥ 0 deny-rules, followed by a final allow-rule. A default-deny firewall is a policy where the final rule is a deny-rule which matches all packets. Similarly, a default-allow firewall is a policy where the final rule is an allow-rule that matches all packets. Additionally, we will use the notation $F(x) = 0$ to denote that a firewall policy F discards a packet x, and $F(x) = 1$ to denote that F accepts the packet x - this notation extends to discard slices as well. Formally, we define a firewall policy F as $F = \{r_1, ..., r_n\}$, where n represents the number of rules, and r_i represents the i^{th} rule in F.

Finally, a "karp-reduction" describes the relationship between languages A and B such that there exists a function f where (i) $x \in A \iff f(x) \in B$, and (ii) the runtime of $f(x)$ is polynomial in time with respect to the input length. This concept is also known as a "many-one" reduction [6], and will be denoted as A $\leq_p$ B for a karp-reduction from language A to B.

3 CONVERTING FIREWALLS TO BOOLEAN FORMULAS

In this section we will describe how TCP packets can be converted into a series of boolean variables, as well as the process of converting a discard slice or firewall policy into a boolean formula. During the processing of a TCP packet by a firewall, the individual values which reside in the packet will be parsed out, and are used to determine if the packet is matched by a policy. To convert this concept into boolean variables and formulas, the integer and IPv4 values which reside in a network packet can be converted into a series of boolean variables, where the number of boolean variables needed to represent a specific value will be dependent on its maximum possible value. Then, the goal is to design a boolean formula such that a TCP packet is accepted by a policy if and only if the boolean variable equivalent of a network packet satisfies the boolean formula representation of the policy. This conversion will be explained in more detail in the following subsections.

3.1 Converting a TCP Packet into Boolean Variables

Consider a TCP packet $pkt = \{(d_1 : v_1), ..., (d_k : v_k)\}$. Each d_i represents the descriptor for a particular field, and each v_i represents it's value in the form of either an integer or an IPv4 address. These values can easily be transformed into a set of boolean variables, in order to create the boolean variable representation of a network packet. Consider the following scheme for converting integers and IPv4 addresses into a series of boolean variables.

(1) Take any integer v_i belonging to a descriptor d_i, where the maximum possible value for a value contained by d_i is depicted as m_i. We can derive the maximum number of bits needed to represent a value belonging to d_i as $bit\_cap_i = ceiling(\log_2(m_i))$. Thus, the integer v_i may be represented by a binary string *bin_str*, where *len(bin_str)* == bit_cap_i. Now, we can create a set of boolean variables *bool_var* which represents v_i using *bin_str*, where $bool\_var[i] = True$ if $bin\_str[i] = 1$, and *False* if $bin\_str[i] = 0$. Following this process, we can create a set of boolean variables which represent v_i.

(2) Consider any IPv4 address v_j. The structure of an IPv4 address is depicted as the joining of four 8-bit integers. Thus, we can simply repeat step (1), once for each 8-bit integer, and concatenate the results together to create a set of 32 boolean variables to represent v_j.

Repeating steps (1) and (2) and concatenating each boolean variable set together, along with using unique variable names for each $(d_i : v_i) \in pkt$, will result in a set of boolean variables which represents pkt. Seeing how we can convert a TCP packet into a boolean variable assignment, we can now move onto converting slices and policies into boolean formulas which can take the boolean representation of a TCP packet as input.

3.2 Considerations Regarding Translating Policies to Boolean Formulas

As explored in Section 2, a firewall rule is a conjunction of integer ranges and/or IP ranges, where if each value present in a TCP packet exists within a range of values defined by the firewall rule, then the packet is said to match that rule, and is decided upon by the firewall using the decision associated with that rule. In order to determine if a TCP packet pkt matches a rule r, then we need to determine if every value within pkt matches it's corresponding field in r. Also stated in Section 2, we know that each value in a TCP packet can be represented by either an integer, or by an IPv4 address. With this in mind, the following questions must be resolved in order to convert a firewall rule into a boolean formula: For an integer range $[a, b]$, can we construct a boolean formula such that the only satisfying variable assignments are the boolean variable representations of integers v where $v \in [a, b]$? Additionally, for a range of IPv4 address $[a, b]$ in CIDR format, can we construct a similar boolean formula that only has sastifying variable assignments for IPv4 address

$v \in [a, b]$? Resolving these questions would allow us to determine if all values within a network packet match a given firewall rule in its entirety.

It should also be noted that the difference between the construction of a general firewall policy and a discard slice are subtle, however the differences in their boolean formula representations can be quite extreme. A general firewall policy has no limitations on the ordered ruleset which defines it, however discard slices with n rules must have the first $n - 1$ rules be allow-rules, and the final rule be a deny-rule. Thus by definition, a discard rule need not care which of the first $n - 1$ rules are matched, as if any of them are matched then the packet will be allowed onto the network, and denied access to the network if it matches the last rule. However general firewall policies have the contingency where only the decision of the first matched rule within the policy will be used to act upon the network packet. Thus each rule of a firewall policy will need to also consider the context of the previous rules before it - if any of the previous rules were matched, then the decision of the current rule or all remaining rules cannot impact the decision already in place.

These concepts are critical in our understanding of the conversion between rules and policies to boolean formulas, and will be explored further in the remaining subsections.

3.3 Converting IPv4 Ranges in CIDR Format to Boolean Formulas

A CIDR range is the typical format for depicting ranges of IPv4 addresses. A CIDR range can be in the form $a.b.c.d/n$, where $a, b, c, d \in [0, 255]$ and $n \in [0, 32]$ - CIDR notation and it's usage is formally defined in RFC 1518 [9]. In order to create a boolean formula to match a particular CIDR range $a.b.c.d/n$, the first n bits of the binary representation of $a.b.c.d$ must be converted into a boolean formula. In the binary representation of the first n bits of $a.b.c.d$, have one variable correspond to one bit. Each of these variables will either be $\neg b_i$ if the i^{th} bit is 0, or b_i if the i^{th} bit is 1, $\forall i \in [0, n]$. Now, the conjugate of all variables will represent a boolean formula which matches all boolean variable represenations of IPv4 addresses within $a.b.c.d/n$. This full algorithm for this operation, `get_ipv4range_formula()`, can be found in Listing 3.

3.4 Converting Integer Ranges to Boolean Formulas

Given an integer range $[a, b]$, as well as bit_cap (the maximum number of bits needed to represent any integer from the descriptor's field), the goal is to convert this into a boolean formula such that only integers represented by boolean variables that are within the range $[a, b]$ will satisfy the formula. The naive solution is to create a formula in disjunctive normal form, where each clause represents an integer in $[a, b]$, and each variable represents a bit in the binary representation of an integer in $[a, b]$. However, for large fields this can become very costly with respect to the size of the formula. Another solution is to recursively split the field into approximate halves, where the midpoint of the recursion corresponds to the most optimal location to create a new clause to represent this integer range. In Listings 1 and 2, the full recursive algorithm which returns the boolean formula for a given integer range, `get_intrange_formula()`, is shown.

3.5 Converting Firewall Rules to Boolean Formulas

It has already been stated that firewall rules are a conjunction of integer and/or IPv4 ranges for each field that is checked by a firewall. Therefore, a firewall rule can be represented by taking each integer range or IPv4 range in a rule, and performing `get_intrange_formula()` or `get_ipv4range_formula()` in order to derive each formula. The boolean formula representation of a firewall rule is then the conjunction of each formula derived from these two algorithms being called on each field that is checked by the firewall. Therefore a TCP packet will match a firewall rule if and only if the boolean formula represenation of the firewall rule is satisfied by the boolean variable representation of the TCP packet. The algorithm for computing a formula for a given firewall rule, `get_rule_formula()`, is explicitly defined in Listing 4.

3.6 Converting Discard Slices and Firewall Policies to Boolean Formulas

Recall that discard slices are a series of allow-rules, followed by a single, final deny-rule. If a packet matches any of the first $n-1$ packets in a discard slice, then the packet is accepted. Thus to create a boolean formula for a discard slice, the first $n-1$ rules of the slice just need to be evaluated. Once this is done, the final formula for the discard slice will be the disjunction of the formula for each of the first $n-1$ rules.

Although as stated earlier, building a boolean formula for an unrestricted policy is trickier than an allow or discard slice. The approach to overcoming this obstacle is detailed in [12], where the researchers were able to build a formula such that each firewall rule is conjugated with another formula that provides context from previous rules. If the current rule being evaluated receives context from previous rules that indicates a previous rule was already matched, then the current rule, as well as the remaining rules, can never be satisfied. Although this allows general firewall policies to be translated into a boolean formula, the resulting formula will grow quadratically as the number of rules increases.

The full algorithms for computing the formulas for discard slices and general firewall policies, `get_ds_formula()` and `get_fw_formula()`, can be found in Listing 5 and Listing 6, respectively.

4 REDUCING FV-D TO UNSAT

In this section, the firewall discard-verification problem, or FV-D, will be formally defined, and shown to be karp-reducable to the boolean unsatisfiability problem, UNSAT. UNSAT is formally defined as the set of all boolean formulas ψ, such that ψ has no satisfying assignment. UNSAT is also the complement to the boolean satisfiability problem (SAT), which is defined as the set of all boolean formulas ϕ such that ϕ has at least one satisfying boolean variable assignment. SAT is known to be NP-complete by reduction from the canonical NP-complete problem [3], which makes the UNSAT problem coNP-complete. In order to understand this reduction, we must first describe the formulation of FV-D instances into boolean formulas, which take the boolean variable representation of a TCP packet as input. We will additionally discuss the number of variables, clauses, and the total size of a given FV-D formula in this section.

4.1 Formally Defining FV-D

FV-D defines the set of discard-slices ds and firewall policies F, such that any packet that is discarded by ds is also discarded by F. To formally define FV-D, have $ds(x)$ and $F(x)$ be the outcome of checking packet x through ds and F, respectively. Then, $\mathit{FV\text{-}D} = \{(ds, F) \mid ds(x) = 0 \implies F(x) = 0\}$. This problem was shown to be "NP-Hard," but more specifically coNP-complete in [4].

In order to solve instances of FV-D, we incorporate the use of a SMT solver, which is detailed in Section 5. SMT solvers have the ability to take a boolean formula as input, or create a boolean formula from some set of data, and use a solver to determine whether or not the formula is satisfiable (i.e. solve SAT instances). Although the use of a SMT solver to resolve UNSAT instances might seem counter-intuitive, we may still use the SMT solvers ability to differentiate between satisfiable and unsatisfiable formulas to determine when a discard slice and firewall tuple belongs to FV-D, and when it does not.

4.2 Formulating FV-D Instances as Boolean Formulas

From the algorithms defined in Section 3, we can define a boolean formula which represents an instance of FV-D. Consider a discard slice ds which has the ruleset $R_{ds} = \{r_1, ..., r_n\}$, where there are n rules in R_{ds}. Additionally, consider the firewall policy F which has the ruleset $R_F = \{s_1, ..., s_m\}$, where there are m rules in R_F. Now, have *ds_formula* = `get_ds_formula(`R_{ds}`)`, and have *fw_formula* = `get_fw_formula(`R_F`)`. We want to determine if for all packets x, $[ds(x) = 0 \implies F(x) = 0] = \mathit{True}$. However we will accomplish this by attempting to show the inverse, in other words, we will attempt to show the existence of a packet where $ds(x) = 0$ and $F(x) = 1$. Proving this relationship exists is equivalent to showing that not all packets denied by ds will be denied by F. So, our resulting formula to describe an instance of FV-D will look like this:

$$\mathit{fvd\_formula} = \neg(\mathit{ds\_formula}) \wedge (fw\_formula)$$

The algorithm for this formulation of a FV-D instance, `get_fvd_formula()`, can be found in Listing 7.

4.3 Reducing FV-D to UNSAT

It's been shown that for a discard slice and firewall policy tuple (ds, F), we can create a corresponding boolean formula. In this subsection, it will be formally proven that both (i) $(ds, F) \in$ FV-D $\iff$ `get_fvd_formula(ds,F)` $\in$ UNSAT; and (ii) that `get_fvd_formula()` is a polynomial-time algorithm with respect to the length of the input (ds, F).

Claim. FV-D $\leq_p$ UNSAT

Proof. First, I will show that the algorithm `get_fvd_formula()` runs in polynomial-time with respect to the size of the input (ds, F). From examining `get_fvd_formula(ds,F)`, it is clear that this algorithm is polynomial-time as long as both `get_ds_formula(ds)` and `get_fw_formula(F)` also run in polynomial-time, with respect to the size of their inputs. Starting with `get_ds_formula(ds)`, it is clear that for each rule $r_i \in ds$, this algorithm also runs in polynomial-time as long as `get_rule_formula(`r_i`)` runs in polynomial-time with respect to the size of r_i. This function then calls `get_ipv4_formula()` and `get_intrange_formula()` a polynomial number of times. `get_ipv4_formula()` only involves a few fixed bitstring operations, and thus only runs in constant time. However, `get_intrange_formula()` is a recursive algorithm that split into halves several times. Specifically, this algorithm splits the integer range being formulated into halves *bitcap* number of times, where $\mathit{bitcap} = \log_2(m_i)$, where m_i is the maximum possible value for a particular field value checked by a firewall. So, the running time of `get_intrange_formula()` is approximately $O(z \cdot \log z)$, where $z = \mathit{bitcap}$, which is polynomial-time. This implies that `get_ds_formula(ds)` is a polynomial-time algorithm with respect to it's input size.

Moving on to `get_fw_formula(F)`, this follows the same convention as `get_ds_formula()`, however there is only an additional polynomial-sized operation between the this function and `get_rule_formula()`. So, this function is also polynomial-time. Considering that `get_ds_formula` and `get_fw_formula()` are both polynomial-time algorithms, it is safe to state that `get_fvd_formula()` also runs in polynomial-time.

Now, it will be shown that $(ds, F) \in$ FV-D $\implies$ `get_fvd_formula(ds,F)`$\in$ UNSAT. In the previous section, it was shown how the conversion between a discard slice and firewall policy tuple, (ds, F), and a boolean formula can occur. Paying attention to how the `get_fvd_formula()` algorithm is designed, it is clear that it preserves the opposite behavior of implication. In other words, if the *fvd_formula* formula (from Section 4.2) remains unsatisfiable, then this implies that no packet denied by ds is allowed by F. Otherwise if it is satisfiable, then the satisfying assignment represents a packet which is denied by ds and allowed by F, showing that the discard implication behavior does not exist. Thus, if $(ds, F) \in$ FV-D, then *fvd_formula* is unsatisfiable, which implies that `get_fvd_formula(ds,F)` $\in$ UNSAT.

Finally, it will be shown that `get_fvd_formula(ds,F)`$\in$ UNSAT $\implies (ds, F) \in$ FV-D. Assume that an unsatisfiable boolean formula exists which was derived from `get_fvd_formula(ds,F)`. This would imply that two firewalls were given as input to produce this formula, and as the formula is unsatisfiable, then these two firewalls must have a discard implication relationship such that $[ds(x) = 0 \implies F(x) = 0] = True$ for all packets x. So, it must follow that since `get_fvd_formula(ds,F)` $\in$ UNSAT, then $(ds, F) \in$ FV-D.

Therefore, `get_fvd_formula()` implements the karp-reduction from FV-D instances to UNSAT instances. In other words, FV-D $\leq_p$ UNSAT. □

4.4 Observations of the Final Boolean Formula

Given we have a structure for what the final FV-D boolean formula looks like, the number of variables, clauses, and the total size of the formula can be approximated. The number of variables in the final formula can be depicted by the sum of the number of bits needed to represent each field in a network packet. For example, if our firewall only checked two IP address and two 16-bit numbers, then the total number of unique variables in the final formula would be 96.

To determine the number of clauses in the final formula, we need to determine the approximate number of clauses at each level of the function call stack for `get_fvd_formula()`. First, we will determine the number of clauses needed to represent the output of `get_ds_formula()`, and then for `get_fw_formula()`, and sum them together. For `get_ds_formula()`, there are $n - 1$ clauses considering if there are n rules, where each clause is a set of another k sub-clauses, and k is the number of field checked by each firewall. Then, if there are k_{ip} number of IPv4 fields checked and k_{int} number of integer fields checked by a firewall (where $k = k_{ip} + k_{int}$), then there are k_{ip} clauses needed for the IPv4 fields, and k_{int} clauses needed to represent the integer fields. Each integer field will also require at most bit_cap_i sub-clauses to represent the i^{th} integer field, where bit_cap_i is the number of bits needed represent any integer in the i^{th} field of the rule. The `get_fw_formula()` algorithm has almost identical setup for m rules, except in addition to the clauses for each rule, the clauses for the previous rules must be appended to provide context for the current rule. Thus, the total number of clauses is approximately equal to:

$$(n-1) \cdot \left(k_{ip} + \sum_{i \in k_{int}} bitcap_i \right) + \left(\frac{m \cdot (m+1)}{2} \right) \cdot \left(k_{ip} + \sum_{i \in k_{int}} bitcap_i \right)$$

The total size of the formula is a similar approximation to the number of clauses, however we also need to account for the number of boolean symbols within each clause. In the case of IPv4 fields, this is a maximum of 32 symbols. In the case of integer ranges, this number is bounded by 2 in the worst case. So, the resulting approximation of the total size of the formula would be bounded by the following total number of variables:

$$(n-1) \cdot \left((32 \cdot k_{ip}) + \sum_{i \in k_{int}} 2 \cdot bitcap_i \right) + \left(\frac{m \cdot (m+1)}{2} \right) \cdot \left((32 \cdot k_{ip}) + \sum_{i \in k_{int}} 2 \cdot bitcap_i \right)$$

Finally, it should be noted that since we are able to formulate FV-D instances into boolean formulas, specifically instances of UNSAT, there are implications towards the use of a SAT/SMT solver to solve instances of these problems. In the following section, the implementation of a SMT solver to resolve FV-D instances will be discussed, and it's performance will be analyzed.

5 *VERIFIRE*

The previous sections of this paper outlined in detail the reduction from instances of FV-D to UNSAT, and deliberated on how this can have implications for SAT/SMT solvers. In order to implement the FV-D $\leq_p$ UNSAT reduction, this paper presents *verifire*, a tool to verify the discard properties of a given discard slice and firewall policy tuple. The remainder of this section details the generation of input test cases and their format, as well as the difference between realistic/feasible and infeasible input sizes by analyzing the running times of several input tests of varying size to *verifire*.

5.1 Input Test Cases

In order to generate test cases, the firewall generation tool *classbench-ng* was used[8] to generate firewall policies which contain an arbitrary amount of rules. A firewall policy can be presented in several formats, however one universally accepted fashion is to have each firewall be represented by a line in a comma-separated-value (CSV) file, where each line contains a series of integer or IPv4 ranges which pertain to a certain field in the network packet. Thus, our tool uses CSV-format discard slices and firewall policies as input. Although the scripts provided by *classbench-ng* output their firewalls in tab-deliminated format, the *verifire* Github repository also hosts a series of scripts that assist in translating *classbench-ng* outputs to CSV format.

By using *classbench-ng* in conjunction with the *bench2csv* scripts from the *verifire* repository, we were able to create a large set of test cases. A subset of them are hosted on the *verifire* repository, with the maximum test case size having 1,000 rules. In the following subsections, we will also discuss how increasing the number of rules per discard slice and firewall policy, up to 5,000 rules, can have an impact on the tools performance.

5.2 Dimensions of Test Cases

In [4], it was stated that the time complexity of solving instances of firewall discard verification is bounded by $O(n \cdot 2^d)$, where n is the number of rules in the policy, and d is the number of bits that are checked by a firewall. The dominating factor here is clearly the 2^d term. This would imply that increasing the number of rules would scale better than increasing the number of bits that are being checked by a firewall. In our testing, we were only able to observe the increase of time to create a formula and time to verify the formula with respect to increasing the number of rules. However, if further testing were done to increase the number of bits checked by a firewall, one would likely see that increasing d would scale worse than increasing n.

5.3 Performance Impact by Large Firewalls

For discard slices and firewall policies where the number of rules is less than 2,000, instances of this problem can be solved in only several seconds, and up to a few minutes. Initially, it can be observed that the time needed to create the FV-D boolean formula takes more time on average than verifying the formula itself. However as the number of rules grows to 3,000 or more firewall rules, we see that the average time needed to verify the formula begins growing exponentially. This can be seen in Figure 1, where we test discard slices and firewall policies of increasing size, iterating by 1,000 rules at a time, up to 5,000 rules. Once *verifire* began testing discard slices and policies with 5,000 rules, the time needed to verify a policy began growing beyond the time needed to create the formula. It can be derived that feasible input sizes, depending on available computing power, can include firewalls up to 3,000 rules, as the combined time needed to create and verify a formula is on average under 10 minutes. Modern day firewall can feature upwards of

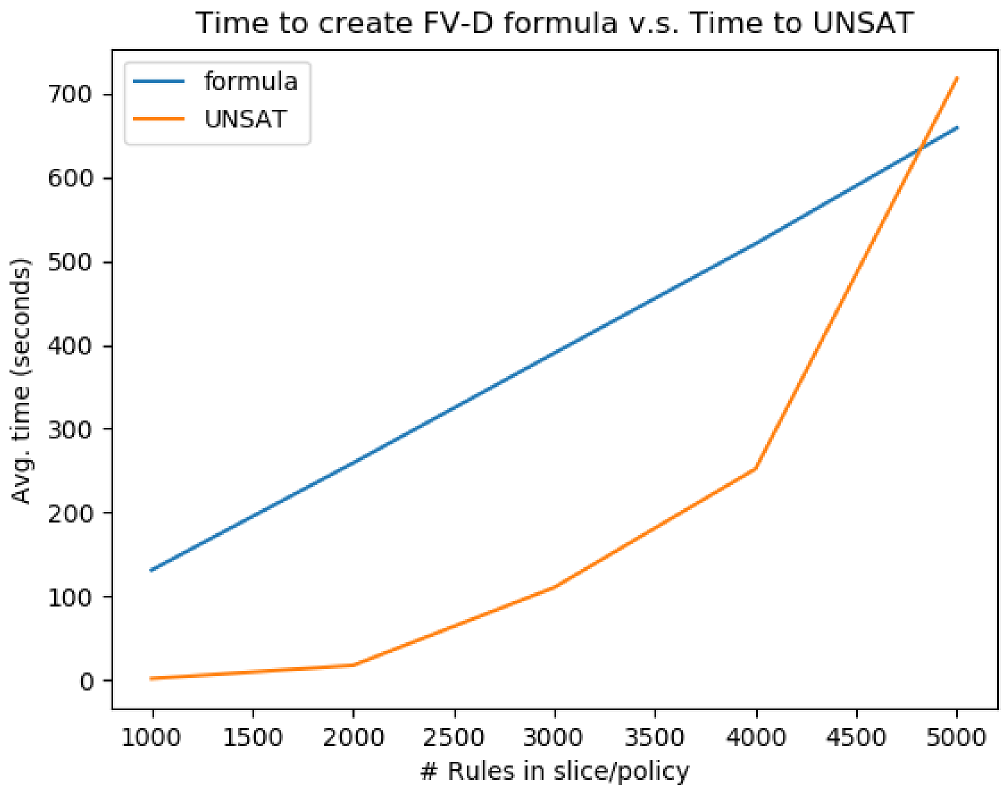

Fig. 2. Time needed to create/verify boolean formula as number of rules increase

a few thousand rules, so inputs of these sizes should be deemed realistic. However as firewall rules go beyond 3,000 rules, the time needed to compute and verify a formula can grow to an infeasible amount.

6 CONCLUSIONS AND FUTURE WORK

In conclusion, this paper has presented the tool *verifire*, which uses a SMT solver to resolve instances of the firewall discard-verification problem. This problem was also shown to be karp-reducable to the boolean unsatisfiability problem, UNSAT. Using test instances derived from *classbench-ng* firewall generators, *verifire* implemented this karp-reduction to confirm whether or not two firewall policies (one of them a discard slice) have a discard implication relationship. Further, through tests performed on discard slices and firewall policies with large rulesets, we were able to show how this verification grows exponentially in time complexity as the number of rules increases. For future works, altering the configuration of *verifire* to accept and generate policies with varying number of fields and varying field sizes would assist in better understanding feasible input sizes. With the growing complexity of cybercrime around the globe, more information pertaining to this problem, less-complex solutions to this problem will always be desired.

REFERENCES

[1] H.B. Acharya and M.G. Gouda. 2010. Projection and Division: Linear-Space Verification of Firewalls. In *2010 IEEE 30th International Conference on Distributed Computing Systems*. 736–743. https://doi.org/10.1109/ICDCS.2010.68

[2] H. B. Acharya and M. G. Gouda. 2011. Firewall verification and redundancy checking are equivalent. In *2011 Proceedings IEEE INFOCOM*. 2123–2128. https://doi.org/10.1109/INFCOM.2011.5935023

[3] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (Shaker Heights, Ohio, USA) *(STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/800157.805047

[4] Ehab S. Elmallah and Mohamed G. Gouda. 2017. Hardness of Firewall Analysis. *IEEE Transactions on Dependable and Secure Computing* 14, 3 (2017), 339–349. https://doi.org/10.1109/TDSC.2015.2455532

[5] Alan Jeffrey and Taghrid Samak. 2009. Model Checking Firewall Policy Configurations. In *2009 IEEE International Symposium on Policies for Distributed Systems and Networks*. 60–67. https://doi.org/10.1109/POLICY.2009.32

[6] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.

[7] Ahmed Khoumsi, Mohamed Erradi, Meryeme Ayache, and Wadie Krombi. 2016. An approach to resolve np-hard problems of firewalls. In *International Conference on Networked Systems*. Springer, 229–243.

[8] Jiří Matoušek, Gianni Antichi, Adam Lučanský, Andrew W. Moore, and Jan Kořenek. 2017. ClassBench-ng: Recasting ClassBench after a Decade of Network Evolution. In *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 204–216. https://doi.org/10.1109/ANCS.2017.33

[9] Yakov Rekhter and Tony Li. 1993. *An architecture for IP address allocation with CIDR.* Technical Report. RFC 1518, september.

[10] Dina Temple-Raston. 2021. *Biden Order To Require New Cybersecurity Standards In Response To SolarWinds Attack*. NPR. https://www.npr.org/2021/04/29/991333036/biden-order-to-require-new-cybersecurity-standards-in-response-to-solarwinds-att

[11] Abi Tyas Tunggal. 2021. *What is the Cost of a Data Breach in 2021?* UpGuard. https://www.upguard.com/blog/cost-of-data-breach

[12] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. 2012. Verification and synthesis of firewalls using SAT and QBF. In *2012 20th IEEE International Conference on Network Protocols (ICNP)*. 1–6. https://doi.org/10.1109/ICNP.2012.6459944

A CODE LISTINGS

A.1 Syntax for Code Listings

The code listings in A.2 are written in pseudocode that is heavily based on Python3 syntax. Any deviations from typical Python3 syntax are listed below, as well as definitions of non-standard functions that aren't significant enough to be given a code listing.

(1) Usage of $\emptyset$, $\neg$, $\vee$ and $\wedge$: The $\emptyset$ symbol is synonymous with the empty array [] object in Python3. Given a formula x, having $\neg x$ denotes the *not* operator, and for two formula x and y, having $x \vee y$ or $x \wedge y$ denotes the *or*/*and* operators, respectfully.
(2) *get_intrange_init(start, end, varname, bit_cap)*: This function will return new values for the *start*/*end* variables in the case that *start* is odd or that *end* is even. A formula will also be returned for any changed value(s).
(3) *int(ceil(log2(x)))*: For a maximum possible field value x, this will return the maximum number of bits needed to represent any integer in the field.
(4) *bin(x,y)*: This will return the binary-string representation of the integer x, where the total length of the bitstring is equal to y.
(5) *count_number_of_zeros(arr)*: This function will return the total number of 0's in an array.
(6) *all_zeros(arr)*, and *all_ones(arr)*: *all_zeros(arr)* function will return *True* if *arr* contains all 0's and False otherwise. *all_ones(arr)* will return *True* if *arr* contains all 1's and False otherwise.
(7) *Symbol(sym_name, Bool)*: This function will return a boolean variable with the name *sym_name*.
(8) *And(form)*, and *Or(form)*: *And(formula)* will combine all symbols/formulas in *form* using the $\wedge$ operator, and *Or(form)* will combine all symbols/formulas in *form* using the $\vee$ operator.
(9) *descriptors* object: This object contains unique variable names for each field, and if the field is an integer range, this object will also contain the largest integer value of the field.
(10) Usage of *rule[decision]*: This is used once in `get_fw_formula()`, and is used to denote the decision of a given rule.

A.2 Code Listings for Reduction from FV-D to UNSAT

```
def get_int_formula(field_val, varname, max_val):
    int_formula = $\emptyset$
    # set int_bitstr to be a bitstring with (bit_cap) number of bits
    bit_cap = int(ceil(log2(max_val)))
    int_bitstr = bin(field_val, bit_cap)
    for b in range(bit_cap):
        sym_name = varname + str(b)
        bit = int_bitstr[b]
        if bit == '1': int_formula += [Symbol(sym_name, Bool)]
        if bit == '0': int_formula += [$\neg$ Symbol(sym_name, Bool)]
    return And(form)
```

Listing 1. Generate Boolean Formula for Single Integer

```
def get_intrange_formula(start_idx, start, end, root, varname, max_val):
    # Base case #1: Initialize starting and ending integers, and intrange_formula
    intrange_formula = ∅
    if end−start == 1:
        return (get_int_formula(start, varname, bit_cap) ∨ (get_int_formula(end, varname, bit_cap)
    elif start_idx == 0 and root == True and (start%2 == 1 or end%2 == 0):
        start, end, init = get_intrange_init(start, end, varname, bit_cap)
        intrange_formula += [init]
    # Find midpoint in current integer range
    bit_cap, current_idx = int(ceil(log2(max_val))), 0
    enum = [bin(value, bit_cap) for value in range(start, end+1)]
    temp_formula = ∅
    for idx in range(start_idx, bit_cap):
        sym_name = varname + str(idx)
        current_idx = idx
        column = [int(line[idx]) for line in enum]
        midpoint = start + count_number_of_zeros(column)
        is_same, element = (all_zeros(column) or all_ones(column)), column[0]
        if is_same and element == 0: temp_formula += [¬ Symbol(sym_name, Bool)]
        if is_same and element == 1: temp_formula += [Symbol(sym_name, Bool)]
        if not is_same: break
    if midpoint%2 == 1 and midpoint−start > end−midpoint: midpoint −= 1
    elif midpoint%2 == 1 and midpoint−start ≤ end−midpoint: midpoint += 1
    # Base case #2: Check if minimal formula exists
    if end−start == 1:
        return And(temp_formula)
    if bin(start, bit_cap)[current_idx:] == '0'*(bit_cap−current_idx) and
    bin(end, bit_cap)[current_idx:] == '1'*(bit_cap−current_idx):
        if temp_formula ≠ ∅: return And(temp)
        else: return None
    # If this point is reached, the [start,end] range needs to be split and recursed at the midpoint
    recurse1 = get_intrange_formula(current_idx, start, midpoint−1, False, varname, bit_cap)
    recurse2 = get_intrange_formula(current_idx, midpoint, end, False, varname, bit_cap)
    recursion = recurse1 ∨ recurse2
    if temp_formula ≠ ∅:
        intrange_formula += [And(temp) ∧ recursion]
    else:
        intrange_formula += [recursion]
    return Or(intrange_formula)
```

Listing 2. Generate Boolean Formula for Integer Ranges

```
def get_ipv4range_formula(ip_addr, cidr_mask, varname):
    ipv4_formula = ∅
    # Convert ip_addr to (cidr_mask)-bit bitstring
    ip_addr_ints = [int(octet) for octet in ip_addr.split(".")]
    ip_addr_bitstr = ''.join([bin(octet) for octet in ip_addr_ints])
    ip_addr_bitstr = ip_addr_bitstr[:(cidr_mask)]
    # Create formula
    For b in range(len(ip_addr_bitstr)):
        bit = ip_addr_bitstr[b]
        sym_name = varname + str(b)
        if bit == '1': ipv4_formula += [Symbol(sym_name, Bool)]
        if bit == '0': ipv4_formula += [¬ Symbol(sym_name, Bool)]
    return And(ipv4_formula)
```

Listing 3. Generate Boolean Formula for IPv4 Ranges

```
def get_rule_formula(rule, descriptors):
    fields = rule.split(",") # split rule into fields
    rule_formula = ∅
    for field in fields:
        varname = descriptors[field]
        if type(field) == "ipv4":
            ip_addr, cidr_mask = field
            field_formula = get_ipv4range_formula(ip_addr, cidr_mask, varname)
        elif type(field) == "int_range":
            max_val = descriptors[field]["max"]
            field_min, field_max = field
            if field_min == field_max:
                field_formula = get_int_formula(field_min, varname, max_val)
            else:
                field_formula = get_intrange_formula(0, field_min, field_max, True, varname, max_val)
        if field_formula ≠ None:
            rule_formula += [field_formula]
    if rule_formula == None:
        return True
    else:
        return And(rule_formula)
```

Listing 4. Generate Boolean Formula for Firewall Rule

```
def get_ds_formula(ds_rules, descriptors):
    ds_formula = $\emptyset$
    for rule in ds_rules:
        rule_formula = get_rule_formula(rule, descriptors)
        ds_formula += [rule_formula]
    return Or(ds_formula)
```

Listing 5. Generate Boolean Formula for Discard Slice

```
def get_fw_formula(fw_rules, descriptors):
    fw_formula = $\emptyset$
    p_i = False
    for rule in fw_rules:
        r_i = get_rule_formula(rule, descriptors)
        m_i = r_i $\wedge$ ($\neg$ p_i)
        if rule[decision] == "allow":
            fw_formula += [m_i]
        p_i = (p_i $\vee$ m_i)
    return Or(fw_formula)
```

Listing 6. Generate Boolean Formula for Firewall Policy

```
def get_fvd_formula(ds_rules, fw_rules, descriptors):
    ds_formula = get_ds_formula(ds_rules, descriptors)
    fw_formula = get_fw_formula(fw_rules, descriptors)
    fvd_formula = $\neg$ ds_formula $\wedge$ fw_formula
    return fvd_formula
```

Listing 7. Generate Boolean Formula for Firewall Discard-Verification Instances