Themis: Ambiguity-Aware Network Intrusion Detection based on Symbolic Model Comparison

Zhongjie Wang
University of California, Riverside
Riverside, CA, USA
zwang048@ucr.edu

Shitong Zhu
University of California, Riverside
Riverside, CA, USA
shitong.zhu@email.ucr.edu

Keyu Man
University of California, Riverside
Riverside, CA, USA
kman001@ucr.edu

Pengxiong Zhu
University of California, Riverside
Riverside, CA, USA
pzhu011@ucr.edu

Yu Hao
University of California, Riverside
Riverside, CA, USA
yhao016@ucr.edu

Zhiyun Qian
University of California, Riverside
Riverside, CA, USA
zhiyunq@cs.ucr.edu

Srikanth V. Krishnamurthy
University of California, Riverside
Riverside, CA, USA
krish@cs.ucr.edu

Tom La Porta
Pennsylvania State University
State College, PA, USA
tlp@cse.psu.edu

Michael J. De Lucia
U.S. Army Research Laboratory
Adelphi, MD, USA
michael.j.delucia2.civ@mail.mil

ABSTRACT

Network intrusion detection systems (NIDS) can be evaded by carefully crafted packets that exploit implementation-level discrepancies between how they are processed on the NIDS and at the endhosts. These discrepancies arise due to the plethora of endhost implementations and evolutions thereof. It is prohibitive to proactively employ a large set of implementations at the NIDS and check incoming packets against all of those. Hence, NIDS typically choose simplified implementations that attempt to approximate and generalize across the different endhost implementations. Unfortunately, this solution is fundamentally flawed since such approximations are bound to have discrepancies with some endhost implementations. In this paper, we develop a lightweight system Themis, which empowers the NIDS in identifying these discrepancies and reactively forking its connection states when any packets with "ambiguities" are encountered. Specifically, Themis incorporates an offline phase in which it extracts models from various popular implementations using symbolic execution. During runtime, it maintains a nondeterministic finite automaton to keep track of the states for each possible implementation. Our extensive evaluations show that Themis is extremely effective and can detect all evasion attacks known to date, while consuming extremely low overhead. En route, we also discovered multiple previously unknown discrepancies that can be exploited to bypass current NIDS.

CCS CONCEPTS

• **Security and privacy → Intrusion detection systems**; • **Software and its engineering** → *Dynamic analysis.*

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8454-4/21/11.
https://doi.org/10.1145/3460120.3484762

KEYWORDS

Network intrusion detection system, symbolic execution, TCP

ACM Reference Format:
Zhongjie Wang, Shitong Zhu, Keyu Man, Pengxiong Zhu, Yu Hao, Zhiyun Qian, Srikanth V. Krishnamurthy, Tom La Porta, and Michael J. De Lucia. 2021. Themis: Ambiguity-Aware Network Intrusion Detection based on Symbolic Model Comparison. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3460120.3484762

1 INTRODUCTION

Today, Network Intrusion Detecion Systems (NIDS) play a vital role in defending against threats emanating from the network. They employ Deep Packet Inspection (DPI) as an underlying technique; with DPI, they reassemble network packets to form upper-layer data streams. The same technique is also extensively applied in various network devices and midddleboxes, such as those used in ISPs' traffic classification, copyright enforcement, etc. [16]. However, NIDS and DPI are known to be inherently vulnerable to evasion attacks that exploit implementation-level discrepancies stemming from ambiguities in network protocol specifications. NIDS will need to interpret network traffic in the same way as endhosts, to derive accurate information. However, different endhosts may run slightly different implementations of the same protocol, while traditional NIDS only incorporate one specific implementation. To be compatible with various endhost implementations, NIDS typically opt for a simplified implementation that over-approximates the behaviors on the endhosts. This in turn, is the underlying cause of the aforementioned discrepancies.

Many studies [4, 19, 22, 25, 30, 40, 41] have shown that an attacker can unilaterally manipulate its packets to trick the NIDS into losing track of its connections, and thus, successfully achieve evasion. In particular, when stateful protocols such as TCP are used, attackers can inject as few as a single packet to desynchronize the NIDS with respect to the current connection state permanently. Recently, crafting techniques for evasion have matured from manual

to automatic strategy generation [2–4, 32, 41], which enables the quick generation of a large number of successful evasion strategies. Therefore, the threats faced by NIDS are increasingly severe.

As defenders, we seek to proactively prevent such potential attacks from happening, instead of reactively patching NIDS. In this paper, we propose a novel framework, Themis, to defend against evasion attacks. As running all possible network protocol implementations on a NIDS can be prohibitively expensive, prior work has proposed to choose a specific implementation for each endhost that it aims to protect [35]. However, due to the diversity of endhost implementations and the challenge in tracking the software versions on all the protected endhosts, such an approach has not been adopted in practice. In this work, we show that it is not necessary to choose between the various implementations. Rather, one can learn the discrepancies among different implementations ahead of time, and fork the connection states on the NIDS when ambiguous packets are received. The NIDS will then analyze the plurality of forked states in parallel, ensuring that one of the connection states will be synchronized with that of the endhost.

In this paper, we focus on TCP, because it is the underlying protocol of most application-layer protocols, and widely targeted in evasion attacks due to its statefulness [3, 4, 41]. To learn the discrepancies between various implementations (especially for TCP with a long history), Themis leverages symbolic execution to automatically extract high-fidelity models from common endhost network protocol implementations. Compared to manually reconstructed models, our models are faithful representations of the actual software running on the endhosts, and are guaranteed to have exactly the same behaviors. Moreover, the extracted models are in the form of high-level SMT (satisfiability modulo theories) formulas; thus, it is easy to use SMT solvers to automatically compare two models to find discrepancies. Upon finding a comprehensive set of discrepancies, we then build an ambiguity-aware NIDS based on nondeterministic finite automata (NFA) that can effectively and simultaneously support multiple different implementations. This approach has several benefits. First, since we go straight to the endhost implementations, we can abandon the existing over-simplified and over-approximated NIDS implementations that have potentially many more discrepancies. Second, with the distilled discrepancies, we no longer need to blindly run many different implementations (some of them may not exhibit any discrepancy) at the same time. In fact, our Themis-enabled NIDS forks its connection states only when ambiguities are encountered and thus, is cost-effective.

The main challenge in applying symbolic execution in practice is its scalability, especially when the goal is to achieve a complete analysis on complex modern software [8], as with the TCP implementation. This is largely due to the nature of the heavyweight analysis of symbolic execution and well-known problems such as path explosion. In our work, we employ several techniques to improve the performance of symbolic execution, without degrading the fidelity of the results. Specifically, we leverage state merging [24] to drastically reduce the cost of symbolic execution and constraint solving (from days to minutes) with domain knowledge of TCP.

In summary, our main contributions in designing and implementing Themis are the following:

- We use symbolic execution to extract high-fidelity models from TCP implementations. We solve the scalability challenge in symbolic execution leveraging state merging without degrading the fidelity of the results. To the best of our knowledge, we are the first to successfully conduct an exhaustive symbolic execution on full-fledged modern TCP implementations.
- We use constraint solving to automatically compare the symbolic models extracted from different versions of Linux kernels, and then summarize the discrepancies between them. We not only reproduce all previously reported discrepancies between modern Linux kernel versions in the past decade, from 3.0 to 5.10, but also discover a few previously unknown subtle discrepancies.
- We design a novel NFA-based NIDS model that accounts for ambiguities and identify them during runtime. This model enables the NIDS to fork its connection states upon encountering a potential ambiguity associated with an incoming packet, to explore all possible ways that an endhost might handle the packet. We demonstrate that with Themis, a NIDS can successfully capture all existing evasion strategies and the new ones presented in this paper, with negligible additional overhead.

2 BACKGROUND

2.1 NIDS Evasion Based on Traffic Manipulation

NIDS are known to be inherently vulnerable to evasion attacks [30], which typically exploit discrepancies between network protocol implementations of the NIDS (e.g., at the IP, TCP and HTTP layers) and those of the endhosts. Stateful protocols like TCP with complex implementations are likely to manifest a larger set of discrepancies. An attacker can send a sequence of specially crafted network packets with a malicious payload, to make the NIDS and the remote host reassemble them into different data streams. The NIDS will see the reassembled data stream without the malicious payload while the remote host will see the reassembled data stream with the malicious payload, and thus, will be subject to attack. Such discrepancies arise largely due to ambiguities in network protocol specifications, as well as evolution of such specifications (e.g., new features added over the years). For example, in TCP implementations based on RFC 5961 [31], RST packets with sequence numbers in the receive window but not equal to the next expected sequence number are no longer accepted; however, older implementations still accept such RST packets. Different operating systems (OSes) and different versions thereof, all differ in their implementations. Even subtle discrepancies have been shown to lead to an evasion attack against NIDS [41]. Importantly, our observation is that NIDS usually use much simpler network protocol implementations as compared to endhosts to reduce their overhead, which widen the gap between their implementations and those of the endhosts.

Researchers have leveraged these discrepancies to design numerous evasion strategies that can bypass state-of-the-art NIDS [4, 25, 40, 41]. For example, if a RST packet is accepted by a NIDS but not by an endhost, the NIDS will consider the connection to be terminated and lose track of the connection. On the other hand, if a RST packet is accepted by an endhost but not a NIDS, the NIDS will keep track of this terminated connection and if a later connection reuses the same 4-tuple, the NIDS will fail to track the new

connection. Similar strategies have been crafted by manipulating control packets such as SYN or FIN packets, as well as data packets. On the defense side, mitigations such as traffic normalization [19] and Active Mapping [35] have been proposed. However, they either cannot eliminate all ambiguities or require additional information from endhosts and thus, still leave opportunities for attackers.

The root cause of the problem is that a traditional NIDS applies a specific network protocol implementation, but there are many different implementations running on the endhosts, all compliant with the same protocol specification. Thus, the NIDS cannot always recover the same information from the network traffic as that by an endhost. Blindly running all different implementations on the NIDS can be prohibitive in terms of overhead. In order to solve this problem, we propose an NFA-based NIDS that forks the connection state only when ambiguities are encountered. Our approach enables NIDS to explore the appropriate possibilities, while introducing relatively low overhead. However, this requires prior knowledge of existing implementations running on the endhosts. To enable this, we employ symbolic execution to extract high-fidelity models from implementations, and empower the NIDS with these models.

2.2 Symbolic Execution and State Merging

Symbolic execution is a formal program verfication technique to systematically find bugs or verify properties in software programs [5, 23]. With promising breakthroughs in automatic reasoning via SAT and SMT solvers, researchers are now widely adopting symbolic execution. Due to its heavyweight analysis, symbolic execution can still only be applied to a small scope of the program, and has to be carefully tuned to avoid uncontrollable path explosions. In addition, practical programs may contain external code not traceable by the symbolic executor, or complex constraints involving non-linear arithmetic or transcendental functions [1]. To make symbolic execution more practical, researchers have proposed "concolic" execution [26], a mixture of concrete execution and symbolic execution, which allows concrete execution to kick in when symbolic execution is incapable or inefficient in dealing with certain parts of the program.

Selective symbolic execution [13] is an innovative form of "concolic" execution that allows switching between symbolic execution and concrete execution at code boundaries. This will restrict symbolic execution only within the scope of interest, while running other parts of the code (e.g., libraries and system calls) with the much faster concrete execution. Defining the boundary between symbolic execution and concrete execution is usually tricky. An exhaustive symbolic execution is theoretically both sound and complete. Here soundness means all inputs derived are guaranteed to yield expected outcomes, i.e., no false positives; completeness means all inputs are covered, i.e., no false negatives. Selective symbolic execution may have impacts on both soundness and completeness while improving performance, because it doesn't completely model all possible outcomes of the code being executed concretely. Therefore, we need to carefully define the scope of symbolic execution to make sure no side effects that may impact the main logic will be introduced by the code out of scope. If any side effects were introduced, we may miss them and be subject to loss of soundness and completeness (by introducing false positives and false negatives). In the Linux kernel, we only run symbolic execution on the TCP core logic, while leaving other parts of the kernel as out of scope.

Prior works [7, 10, 41] have used symbolic execution to verify properties or discover bugs in programs. However, they randomly explore only parts of the program, and get partial coverage. These approaches aim at opportunistically finding bugs rather than achieving complete coverage. This causes loss of both completeness and soundness, and leads to false negatives and false positives. Differently, our goal is to extract a complete model of the target code we are interested in, so that we can retain completeness and soundness, which means no false negatives or false positives. As discussed, scalability is known to be the biggest challenge in symbolic execution. The problem worsens when running symbolic execution on binaries rather than on source codes, since more branches could be introduced into the low-level assembly code after compilation.

To achieve scalability, researchers have proposed state merging [20], which can reduce the number of execution paths in symbolic execution, but at the cost of introducing harder-to-solve constraints for the constraint solver. We use an example in Listing 1 to illustrate the rationale of state merging. A symbolic execution state is defined as a 3-tuple (ℓ, σ, π). ℓ denotes the current program location; σ denotes the symbolic store that stores all symbolic and concrete values associated with the current state; π denotes the path constraints. In Figure 1 and Figure 2, we demonstrate the process of symbolic execution without and with state merging respectively. Without state merging, a state forks when a conditional branch is encountered and both branches are feasible. The number of states doubles each time and so, there will be 4 states after two branches. With state merging, two states at the same ℓ can be merged by: 1) combining their paths constraints π with a logical OR; 2) merging their symbolic stores σ with if-then-else (ITE) expressions. For example, when two states $(\ell : 8, \sigma : a = 5, \pi : x > 10)$ and $(\ell : 8, \sigma : a = -5, \pi : x \leq 10)$ meet at ℓ 8, their paths constraints are merged into $x > 10 \vee x \leq 10$, which can be simplified to $true$, and the value of variable a becomes $ite(x > 10, 5, -5)$.

```
1  int foo(int x, int y) {
2    int a = 0;
3    if (x > 10) {
4      a = 5;
5    } else {
6      a = -5;
7    }
8    if (y == 1) {
9      ++a;
10   } else {
11     --a;
12   }
13   return a;
14 }
```

Listing 1: Sample code snippet for state merging

After two rounds of state merging, we have only one state. By comparing the results with and without state merging, we find that in the latter case, there are an exponential number of states, concrete values for variable a, and complex path constraints. In contrast, with state merging, there are much fewer states, much simpler path constraints, but complex expressions for symbolic variables. Because ITE expressions introduce more complex expressions that

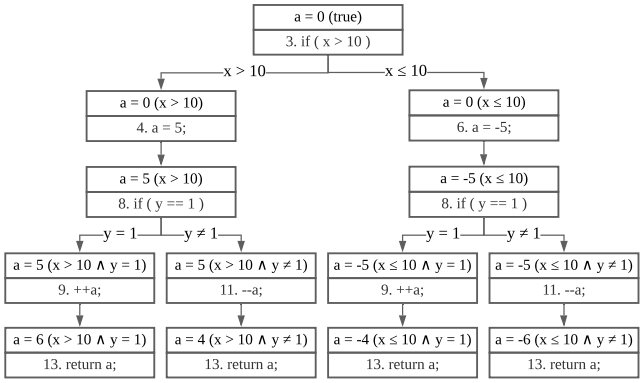

Figure 1: Symbolic execution without state merging

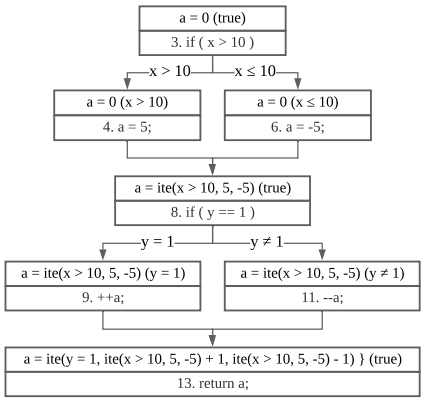

Figure 2: Symbolic execution with state merging

are translated into disjunctions, they may cause significant overhead in constraint solving, and eventually negate the benefits from state merging [20]. Essentially, we are shifting the burden from the symbolic executor to the constraint solver, and thereby need a good balance. Kuznetsov et al. [24] provide insights into the problem and show that two states should be merged if the variables they differ in, are less frequently used in later queries to the constraint solver.

3 OFFLINE PHASE: DISCOVERING TCP IMPLEMENTATION DISCREPANCIES

The offline phase of Themis that finds discrepancies between any two TCP implementations has three key components, as shown in Figure 3. The first component is *Symbolic Model Extraction*, which runs symbolic execution exhaustively on different versions of TCP implementations and extracts high-fidelity models to accurately reflect detailed behaviors of each implementation. The second is called *Model Comparison*, which compares two symbolic models and automatically generates concrete examples that will trigger the discrepancies between them. The last is *Discrepancy Analysis*, which empirically analyzes the execution traces corresponding to the concrete examples and determines the root cause of a discrepancy. The process is iterative in that we feed the discrepancies summarized from *Discrepancy Analyis* back to the *Model Comparison* to exclude them from the models in the next round of concrete example generation, until there are no discrepancies between the two models. The discrepancies discovered will be integrated into the NIDS to enable online operations of Themis as discussed in §4.

3.1 Symbolic Model Extraction

Finding low-level discrepancies between two TCP implementations is a daunting task, because of the huge number of possible states in the program. Such discrepancies could be buried deep, in some rarely visited states. To formalize, discrepancies occur when two implementations generate different outputs given the same input. With respect to the NIDS evasion attacks, a discrepancy occurs when two TCP implementations produce different reassembled data streams, when they receive the same sequence of TCP packets. To aid the discovery of discrepancies, we first define a set of critical states S as intermediate states that precede our target output (i.e., the reassembled data stream). We consider two types of critical states as follows: (a) the TCP states, e.g., LISTEN, SYN_RECV, ESTABLISHED, CLOSE, and (b) receive buffer events, e.g., whether a packet enters the TCP in-order queue or out-of-order queue. Intuitively, if the same sequence of input packets drives two TCP implementations into two different TCP states or into accepting different payloads in the TCP receive buffers, such discrepancies are bound to be exploitable.

As shown in Figure 4, via exhaustive symbolic execution, we extract a mapping M between the path constraints Π and the critical states S, denoted as $M : \Pi \rightarrow S$. Since path constraints are constraints on the inputs, this translates to summarizing the relationships between inputs and critical states. By combining all the path constraints that lead to a critical state with disjunction, we can automatically obtain the *weakest precondition* [15] of the critical state. The weakest precondition, denoted as $wp(S, R)$, is the condition that characterizes all possible initial states making a system S terminate in a final state that establishes the truth of an assertion (post-condition) R. The term "weak" or "strong" allude to how general or specific a condition is. The weakest precondition is basically the most general constraints that should be satisfied in order to satisfy a given postcondition. Weakest precondition is commonly used in the generation of verification conditions [15]. In our setting, the postconditions are the critical states that we label. Therefore, we have the following equation, in which I denotes the TCP implementation:

$$wp(I, s) = \bigvee_{M(\pi_i)=s} \pi_i \tag{1}$$

To provide a concrete example of a discrepancy leading to differing path constraints in different versions of the TCP implementation, we show in Listing 2 and Listing 3 how Linux validates incoming RST packets in different versions. In Linux kernel versions before 3.6, when in the ESTABLISHED state, it accepts a RST packet as long as its sequence number is within the current receive window (Line 6); it then resets the connection and enters the CLOSE state (Line 13). In versions after 3.6, Linux developers implemented the defense mechnism from RFC 5961 [31], which performs a much stricter check on RST packets. These versions only accept a RST packet if its sequence number exactly matches the

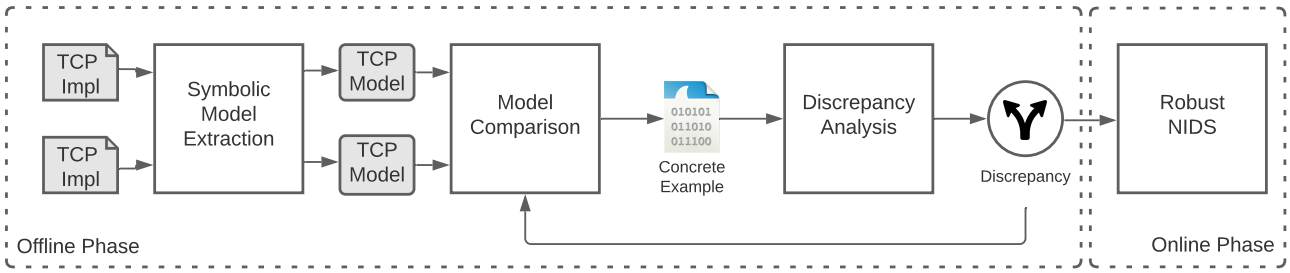

Figure 3: System Overview of Themis

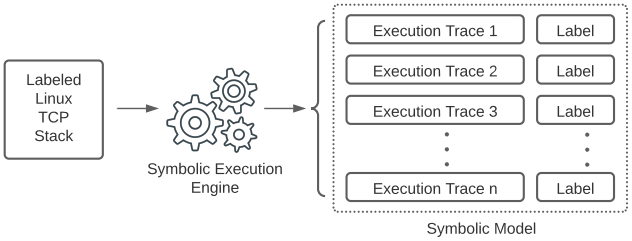

Figure 4: Symbolic Model Extraction

next expected sequence number *rcv_nxt* (Line 19). In this case, the two implementations differ in the path constraints relating to the CLOSE state. In the earlier versions, the differing path constraints include $rcv\_nxt < seq\_num < rcv\_nxt + window\_size$; in the latter versions, the differing path constraints include $seq\_num = rcv\_nxt$.

```
1  static int tcp_validate_incoming(struct sock *sk,
       struct sk_buff *skb, struct tcphdr *th, int
       syn_inerr)
2  {
3      ...
4      /* Step 1: check sequence number */
5      if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq,
       TCP_SKB_CB(skb)->end_seq)) {
6          ...
7          goto discard;
8      }
9      /* Step 2: check RST bit */
10     if (th->rst) {
11         tcp_reset(sk);
12         goto discard;
13     }
14     ...
15 }
```

Listing 2: Validation of RST packets in Linux kernel versions before 3.6

```
1  static bool tcp_validate_incoming(struct sock *sk,
       struct sk_buff *skb, const struct tcphdr *th,
       int syn_inerr)
2  {
3      ...
4      /* Step 1: check sequence number */
5      if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq,
       TCP_SKB_CB(skb)->end_seq)) {
6          ...
7          goto discard;
8      }
9      /* Step 2: check RST bit */
10     if (th->rst) {
11         if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt)
12             tcp_reset(sk);
13         else
14             tcp_send_challenge_ack(sk, skb);
15         goto discard;
16     }
17     ...
18 }
```

Listing 3: Validation of RST packets in Linux kernel versions after 3.6

3.1.1 Eliminating Non-determinism in TCP Processing. To extract a deterministic model from a TCP implementation, we need to first eliminate any non-determinism in TCP. Non-determinism can cause symbolic execution to explore different parts of the code in different runs, and thus causes false positives, i.e., "discrepancies" found between non-deterministic models may not exist between the actual implementations. We fix the order of symbolic path exploration, run symbolic execution multiple times, and then compare the differences in path coverage in order to find any non-determinism that might exist. Note that non-determinism is always introduced by concrete inputs, because symbolic inputs will enable forking in symbolic execution and exploration of all feasible paths. One example of such concrete inputs is a random number generated during execution, e.g., the initial sequence number in TCP. Since we model the server-side logic of a TCP implementation, we assume the client-side sequence number is controllable by the attacker and thus symbolize it. Meanwhile, we hook the random number generator for the server-side initial sequence number and coerce it to always return a fixed number to eliminate non-determinism.

Other non-determinism may be introduced due to variations in the execution times of symbolic execution; this would influence factors such as (but not limited to): 1) Timeouts (e.g., TCP connection timeout, packet transmission timeout); 2) Congestion control window size computations; 3) Round-trip time (RTT) calculations; 4) Receive buffer size computations; 5) Delayed ACK computations; 6) MTU probing; 7) Rate-limits (e.g., out-of-window ACKs, challenge ACKs); 8) Socket locking by the user thread (affected by the timing of kernel and user thread switching). To eliminate the non-determinism introduced by the variation of execution time, we hook the TCP access to the system clock and always return deterministic

values. This could potentially lead to reduced code coverage (e.g., no timeouts). We argue that this is a reasonable decision because even if a discrepancy exists in such timing-related code blocks, it can be unreliable to use such a discrepancy to perform an evasion attack. In fact, we have not seen any report of such discrepancies leveraged to that effect. In our experiments, we freeze the clock by always returning the same exact value.

3.1.2 State Merging to Achieve Scalability. To handle path explosion in symbolic execution, we adopt the idea of state merging from [24]. There is a gamut of state merging options on a program, from complete separation of individual execution traces (no merging) to aggressively merging two states whenever their execution traces join (static state merging). As discussed in §2, state merging reduces repetitive work of executing the same code blocks in symbolic execution at the cost of introducing harder-to-solve formulas in constraint solving. Specifically, if a variable holds two different values (either concrete or symbolic) in two states, after merging, the value of the variable will become an ITE expression, which will increase the burden of the constraint solver. Too aggressive state merging may even harm performance rather than improve it [20]. Hence, we employ state merging following the general suggestions from [24] as well as the domain knowledge of TCP.

Specifically, we first collect a list of fork points during an initial run of symbolic execution. Then we mark *merge range candidates* with the fork points as the starting points and their immediate post-dominators as the ending points. The start and end points form candidate merge ranges. After that, we manually inspect each candidate and the variables being modified within it, and decide whether to label it as a merge range based on the following heuristics: 1) the critical state variables should not be modified within the merge range; otherwise, paths belonging to different critical states may get merged, and it becomes complicated to group execution traces by critical states in the later phase; 2) no dynamic memory allocation or deallocation should occur within the candidate range; otherwise, pointers may become symbolic after merging, which will induce complexity and extra overheads; this also includes sending packets, because it will dynamically allocate new buffers for the packets to be sent; 3) no excessive number of variables modified within the candidate range (especially if there are TCP-related state variables that will be used heavily subsequently); otherwise, extra complexity will be introduced in constraint solving later. Note that these heuristics can be potentially automatically applied with the help of static analysis. Nevertheless, we consider it an orthogonal component which can be improved upon separately. We leave the automation of the merge range determination as a future work.

Finally, during the actual symbolic execution, the labelled merge ranges are applied accordingly. If the merge ranges are nested, we first merge the innermost ranges, and then the outer ones.

3.2 Model Comparison

The symbolic model extracted in the previous step is in the form of a mapping from path constraints on inputs, to critical states, as shown in Figure 4. Here the critical states can be considered as the intermediate states that are directly related to the output state, which is the reassembled data stream that will be passed to the application layer. Instead of finding the differences in critical states given the same input, we try to find the differences in inputs given the same critical state. Specifically, we group the exectuion traces by critcal states, and then combine their path constraints with disjunction. The combined path constraints reflect all possible inputs that will drive the TCP implementation into the critical state, which is also equivalent to the weakest precondition of the critical state. Then we compare the combined path constraints from two different TCP implementations, for each of the critical states, as shown in Figure 5. Note that the path constraints are represented in a format (SMT-LIB [37]) that can be directly processed by constraint solvers; thus, we can automatically test if two path constraints are equivalent using state-of-the-art constraint solvers such as Z3 [42]. The constraint solver will either prove that the two path constraints are equivalent or generate a concrete counterexample that is accepted by one of the path constraints but not the other.

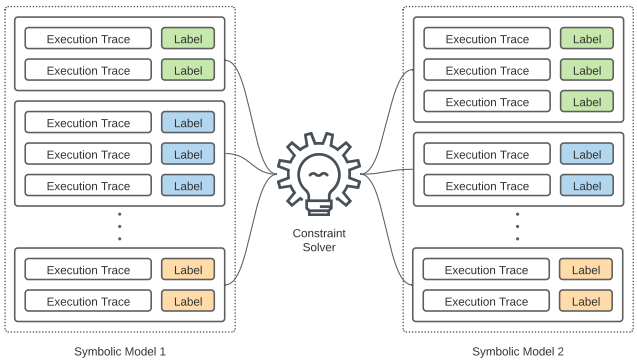

Figure 5: Symbolic Model Comparison

3.3 Discrepancy Analysis

From the counterexample generated from the last step, we can craft TCP packets that will trigger the discrepancy between the two TCP implementations. However, our goal is to learn a class of packets that belong to a specific discrepancy (e.g., RST packets with in-window sequence number) and then, summarize the discrepancy in general. To this end, we first feed the TCP packets generated from a counterexample, to both TCP implementations and record the execution traces, respectively. Then, we manually reason about the root cause of the difference in their execution traces. Since the execution traces are from two different implementations, we cannot directly compare them to find the discrepancy. Further, they are at the binary level and little information is provided. To ease the analysis, we translate the binary-level execution traces to source-code-level execution traces. From there, we can focus on the symbolic branch traces and easily identify the differences and the critical branches. Subsequently, we summarize the difference into a symbolic formula, which can be used to identify the discrepancy. The symbolic formula is fed back to the model comparison phase, to exclude discrepancies that have already been found; so we can iteratively discover new discrepancies until none exist. To exclude a discrepancy from a model, we need to perform a conjunction with the negation of the symbolic constraint of the discrepancy on the original path constraints. The algorithm is shown in Algorithm 1.

Algorithm 1 Finding discrepancies between two implementations

```
1: function FindAllDiscrepancies(I1, I2)
2:     AllDiscrepancies ← ∅
3:     M1 ← ExtractSymbolicModel(I1)
4:     M2 ← ExtractSymbolicModel(I2)
5:     for s ∈ all critical states in M1 or M2 do
6:         Discrepancies ← CompareSymbolicModels(M1, M2, s)
7:         AllDiscrepancies = AllDiscrepancies ∪ Discrepancies
8:     end for
9:     return AllDiscrepancies
10: end function
11: function CompareSymbolicModels(I1, I2, M1, M2, s)
12:     Discrepancies ← ∅
13:     Π1 ← ⋁_{M1[π]=s} π
14:     Π2 ← ⋁_{M2[π]=s} π
15:     Result, CounterExample ← SolveConstraints(Π1 = Π2)
16:     while Result = unsat do
17:         Discrepancy ← DiscrepancyAnalysis(I1, I2, CounterExample)
18:         Discrepancies.insert(Discrepancy)
19:         Π1 ← Π1\Discrepancy.constraints
20:         Π2 ← Π2\Discrepancy.constraints
21:         Result, CounterExample ← SolveConstraints(Π1 = Π2)
22:     end while
23:     return Discrepancies
24: end function
25: function DiscrepancyAnalysis(I1, I2, CounterExample)
26:     ExecTrace1 ← TraceExecution(I1, CounterExample)
27:     ExecTrace2 ← TraceExecution(I2, CounterExample)
28:     Discrepancy ← RootCauseAnalysis(ExecTrace1, ExecTrace2)
29:     return Discrepancy
30: end function
```

An alternative workflow is to find a counterexample, exclude the path constraints corresponding to that, from both symbolic models, and then find the next counterexample. This would decouple the Symbolic Model Comparison from the Discrepancy Analysis, and make the former fully automated. However, there can be a large number of execution traces corresponding a single discrepancy, and so it could take much longer to exclude a discrepancy than using the feedback from the Discrepancy Analysis directly (our initial studies indicate this is the case). Besides, an execution trace may cover more than one discrepancy and thus the exclusion of an entire trace may remove more than the current discrepancy. Thus, we did not pursue this second approach in our work.

3.4 Finding Discrepancies in Linux TCP Stack

In this work, we seek to find discrepancies in the TCP stack between different versions of the Linux kernel, but our method could be applied to other OSes as well. Suppose we choose n Linux kernel versions to analyze, and sort them by version numbers. We will compare each pair of adjacent Linux kernel versions using the approach described earlier, and summarize the discovered discrepancies. Ideally, we should compare every pair of versions to find all discrepancies that exist among them; however, since the Linux versions are ordered, we argue that it is sufficient to just compare adjacent versions and still get the same result. Assuming there are three versions in order, v_1, v_2, and v_3, if there is no discrepancy between v_1 and v_2, nor between v_2 and v_3, then it is unlikely that there will be a discrepancy between v_1 and v_3. We give a proof by contradiction as follows. If there was a discrepancy between v_1 and v_3, then there must be some difference in the path constraints relating to some critical state, between v_1 and v_3. This difference must have been introduced either between v_1 and v_2, or between v_2 and v_3. Therefore, we must be able to catch that, either when comparing v_1 and v_2, or when comparing v_2 and v_3.

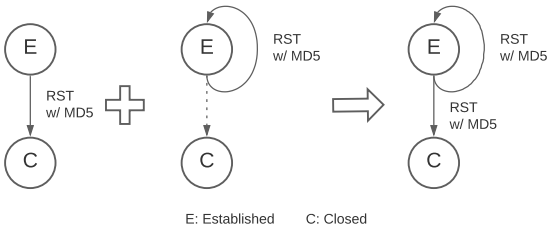

Figure 6: Merging DFAs into NFA

4 ONLINE PHASE: AMBIGUITY-AWARE NIDS

4.1 NFA-Based Model for NIDS

Stateful network protocols are usually modeled as Deterministic Finite Automata (DFA) to make sure that different parties associated have deterministic behaviors and are well-synchronized. NIDS also use DFA-based network protocol implementations (as do the end-hosts). However, this makes them conform to a specific version of a protocol implementation, and may have discrepancies with other versions. This leaves opportunities for attackers to evade them. In order to ensure compatibility with different versions of the network protocol implementation, we propose a novel, NFA-based model for NIDS. In Nondeterministic Finite Automata (NFA), upon receiving an input, the state could non-deterministically transition into any of multiple different new states. If there is any possibility that the state transitions into an accepting state, the input is accepted. In order to capture every possibility, an NFA needs to "clone" its state when there are multiple possible state transitions [36]. Thus, the NFA-based model enables the NIDS to handle packets with ambiguities i.e., compatible with different packet handling logics of different versions.

Discrepancies across TCP implementations project ambiguities to the NIDS when handling packets. We define an ambiguity as a 3-tuple, $(\varphi, \lambda_1, \lambda_2)$, derived from the discrepancies found by Themis as discussed in §3. φ denotes the symbolic formula characterizing all possible inputs that trigger the behavioral differences between two implementations. λ_1 and λ_2 denote the differing behaviors of the two respective implementations. We integrate the ambiguities into the existing DFA of the NIDS and turn it into an NFA. φ is the guard or predicate of the transition. There are two output states, and λ_1 and λ_2 are the corresponding transition functions. In this way, we are merging multiple DFAs into an NFA while reusing the common parts in the DFAs to the maximum extent possible. An example is shown in Figure 6. When processing a RST packet with a TCP MD5 option, an earlier version of Linux accepts it while a later version discards it. After merging the two DFAs, the NFA will explore both possibilities in parallel. Rather than running two DFAs side by side, Themis allows maximized reusability of the code and expedites packet processing.

When processing a packet that causes an ambiguity, the NFA-based NIDS will "fork" its currently maintained state for the TCP connection and process it with λ_1 and λ_2 respectively. Note that for each connection and ambiguity, we only fork once and remember

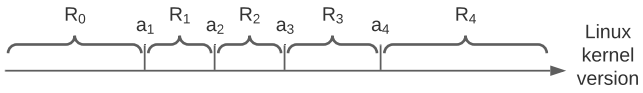

Figure 7: Example: version ranges defined by ambiguities

the behavior associated with each copy of the connection state. Assuming there are k ambiguities, then the upper bound of the number of connection state copies is 2^k. If the attacker is also aware of the ambiguities, then he can intentionally inject them into his traffic and cause an exponential growth in the number of connection states on the NIDS, as a resource exhaustion attack. As a further optimization, we take version coherence of the behaviors into account and reduce the growth rate to a linear rate.

4.2 Version Coherence

Since ambiguities are introduced in certain versions as code changes, versions between two adjacent ambiguities have the same behaviors with regard to all ambiguities. Thus we only need to maintain one connection state for those versions in a range. We divide the version space of a particular implementation into version ranges by ambiguities. Given that the ambiguities are found from comparing adjacent versions (mentioned in §3.4), the ambiguities are naturally ordered as well. For example, if we find 4 ambiguities through our offline symbolic model comparison, and sort them by the versions in which they were introduced, say $a_1 < a_2 < a_3 < a_4$, they will divide the version space into 5 ranges, R_0, R_1, R_2, R_3, and R_4, as shown in Figure 7. Each range corresponds to a set of TCP behaviors that are consistent with regard to all ambiguities. For example, R_1 represents the behaviors consistent with the new behavior of a_1 and the old behavior of a_2, a_3, a_4.

Our observation is that given a specific TCP implementation, its behavior must fall under one of the ranges. Therefore, following the above example, in the worst case, we should only need to maintain at most 5 connection states, which is $k+1$, as opposed to 2^k. To see this, consider that the NIDS first encounters ambiguity a_1 during the online phase; it will then fork into two connection states s_1 and s_2, where s_1 represents the range R_0, and s_2 represents the range from R_1 to R_4. If the NIDS then encounters a_2, s_2 will then be forked into s_3 and s_4, representing the range R_1, and R_2 to R_4. It is important to note that s_1 is not forked at this step because the newly discovered ambiguity a_2 is irrelevant to that state. In other words, the NIDS already accounts for any server behaviors that are consistent with R_0, and we assume the server's behavior cannot change arbitrarily to those consistent with R_2, R_3, or R_4 suddenly (the server is not malicious). From hereon, it is not hard to see that if the NIDS discovers a_3 and a_4 subsequently, we will end up with exactly 5 connection states.

Note that there is a special case where multiple ambiguities may relate to the same kind of packets, i.e, they have overlapping input spaces φ. For example, a RST packet with sequence number equals to the end of the rightmost SACK block is also an in-window RST packet. It corresponds to both Discrepancy 3 and 7 in Table 2. In this case, each version range should follow the behavior of the nearest ambiguity if conflicting behaviors are encountered. Upon receiving such a packet, the NIDS will fork twice and creates three version ranges. Because the old behaviors of Discrepancy 3 and 7 are conflicting, so the version range before Discrepancy 3 should follow the old behavior of Discrepancy 3, rather than Discrepancy 7. Similarly, the version range after Discrepancy 7 should follow the new behavior of Discrepancy 7, rather than Discrepancy 3.

In Algorithm 2, we show the algorithm used in Themis for handling packets with ambiguities in the NIDS.

Algorithm 2 Handling packets with ambiguities in NIDS

```
1: procedure OnPacketReceived(Packet)
2:     NewConnStates ← ∅
3:     for ConnState ∈ AllConnStates do
4:         ConnState.Ambiguities ← CheckAmbiguities(Packet, ConnState)
5:         for AmbiguityID ∈ ConnState.Ambiguities do
6:             if ConnState.Behaviors[AmbiguityID] = Undefined then
7:                 NewConnState ← Fork(ConnState)
8:                 for i ∈ [AmbiguityID, MaxAmbiguityID) do
9:                     NewConnState.Behaviors[i] ← Old
10:                end for
11:                for i ∈ [0, AmbiguityID) do
12:                    ConnState.Behaviors[i] ← New
13:                end for
14:                NewConnStates.insert(NewConnState)
15:            end if
16:        end for
17:    end for
18:    AllConnStates ← AllConnStates ∪ NewConnStates
19:    for ConnState ∈ AllConnStates do
20:        HandlePacketWithAmbiguities(Packet, ConnState)
21:    end for
22: end procedure
23: procedure HandlePacketWithAmbiguities(Packet, ConnState)
24:    for AmbiguityID ∈ ConnState.Ambiguities do
25:        Skip if conflicting ambiguities occur and AmbiguityID is not the closest
26:        if ConnState.Behaviors[AmbiguityID] = Old then
27:            Implementation of the old behavior
28:        else if ConnState.Behaviors[AmbiguityID] = New then
29:            Implementation of the new behavior
30:        end if
31:    end for
32:    Processing packets without ambiguities
33: end procedure
```

5 EVALUATION

In this section, we first evaluate Themis's symbolic-execution-based discrepancy discovery, and list the discrepancies found between different versions of Linux. Next, we evaluate Themis's augmented NIDS by integrating all the discrepancies discovered, and show its (1) effectiveness in defending against all existing and newly discovered evasion attacks and (2) performance overhead at runtime.

5.1 Symbolic-execution-based Discrepancy Discovery

Themis's offline components described in §3, are built upon S2E [13] and Z3 [42]. We implement (a) our TCP symbolic execution as S2E plugins, and (b) model comparison and discrepancy analysis with Python scripts using Z3. We run Themis on a machine with an AMD EPYC 7542 Processor (2.9GHz, 32-core, 64-thread), and run symbolic execution with 64 processes in parallel.

5.1.1 Performance of Symbolic Execution and Model Comparison. In our exhaustive symbolic execution, we bound the input to 3 symbolic TCP packets with TCP options and payloads, and run until all execution paths finish. We symbolize the TCP header fields

except the TCP checksum. This can cover all server-side TCP states, including LISTEN, SYN_RECV, NEW_SYN_RECV, ESTABLISHED, CLOSE_WAIT, CLOSE, and critical states related to receive buffer (e.g., receipt of in-order data or out-of-order data). Note that there are other TCP states requiring the endhost to actively initiate or close a connection. Since we are modeling the servers' behaviors, we leave the exploration of those TCP states to future work.

In our initial attempt of exhaustive symbolic execution without state merging on the Linux TCP stack (on version 4.4), we send 3 symbolic packets without any TCP options; it took 13.5 hours on average to finish. The total number of execution paths is 1,219,938. After enabling state merging, it takes less than 4 minutes to finish, and the total number of execution paths decreases to 1386. Using the heuristics provided in §3, we labeled 24 merge ranges. They are mainly related to ECN (Explicit Congestion Notification), window size, MSS (Maximum Segment Size), and urgent pointer. It takes us on average, one day to do the manual labeling for a Linux kernel version. In addition, without state merging, it takes more than a week to compare the rather huge models (with more than one million paths) extracted from Linux kernel versions 4.4 and 5.4. Instead, it takes only 15 seconds to compare two models (with about 1000∼2000 paths) after enabling state merging.

5.1.2 Themis's Discrepancy Discovery. We analyzed 5 major LTS Linux kernel versions spanning over the past decade, viz., 3.0, 3.10, 4.4, 5.4, and 5.10. For each version, we run exhaustive symbolic execution with state merging and send 3 symbolic packets, with and without TCP options. Since TCP options are usually not correlated with each other, we try each TCP option individually instead of combining them. We also symbolize the values in each TCP option. For the experimental results without TCP options, the numbers of execution paths relating to each critical state are shown in Table 1. Note that, these numbers are based on merged execution paths, and are affected by the labeled merge ranges.

We list all the discrepancies found by Themis in Table 2. We also validate our findings with the commit history of the Linux kernel, and note the date and version when a discrepancy was first introduced. As one can see, most of the discrepancies were introduced around 2012, while some newer ones were introduced around 2017. A major reason contributing to these discrepancies, is the change proposed in RFC 5961 [31], a mitigation against blind in-window attacks. The RFC introduces stricter checks on the sequence and acknowledgment numbers in SYN, RST and data packets. This leads to the Discrepancies 2, 3, and 4. A second reason is buggy implementations when validating a TCP packet. Discrepancy 1 is caused by the older versions not doing a propoer validation on TCP flags in the LISTEN state, and accepting invalid TCP flag combinations, i.e., SYN+FIN. Discrepancy 5 is due to older versions not checking ACK flags when processing data packets. Discrepancy 9 is due to older versions mistakenly bypassing the acknowledgment number checking in certain states, e.g., CLOSE_WAIT, CLOSING, LAST_ACK. Other reasons stem from performance improvements and compatibility with other OSes. Discrepancy 6 was introduced by a fix to the implementation of the Fast Retransmit/Fast Recovery algorithm. Discrepancy 7 was introduced for performance optimization when SACK is enabled and packet loss happens frequently. Discrepancy 8 was introduced to handle an idiosyncrasy associated with Mac OSX clients, which may leave a connection that is supposed to be closed, in a lingering state.

Table 1: Number of execution paths grouped by critical states in different versions of Linux kernel

Version	All	SR	EST	CW	CL	IO	OOO
3.0	2966	2841	919	588	84	372	1454
3.10	2344	2128	574	350	40	284	796
4.4	1386	1170	302	181	20	149	410
5.4	2214	1998	729	650	49	428	1140
5.10	2314	2250	863	678	51	440	1322

Notation: SR - SYN_RECV/NEW_SYN_RECV; EST - ESTABLISHED; CW - CLOSE_WAIT; CL - CLOSE; IO - In-order data; OOO - Out-of-order data. The results are accquired with state merging enabled.

We manually inspected the changes to the TCP stack in Linux from version 3.0 to 5.10, and confirmed that the discrepancies listed in Table 2 hold true, and no other discrepancies are found. In addition, we measured these discrepancies on Alexa's top 1 million websites from the client side, by sending probe packets to the servers and collecting responses. We find that both old and new behaviors are observed for all discrepancies, which means today's NIDS can only either incorporate the older or the newer version but not both. This leaves the remaining servers vulnerable to attacks.

In addition to implementation-level discrepancies, we also found some differences in default configurations. Although the TCP MD5 option was introduced in version 2.6.20 in 2006, it was an experimental feature and by default disabled until version 3.9 in 2013. The initial window sizes are different between versions 4.4 and 5.4; this is caused by different configuration values of the TCP receive buffer size, i.e., net.ipv4.tcp_rmem. The default values for minimum, default, and maximum size of the TCP receive buffer are (4096, 87380, 1887552) in version 4.4, but are (4096, 131072, 1772832) in version 5.4, as a result of an increasing demand in throughput.

5.1.3 Case Studies. In this section, we choose three of the newly discovered discrepancies from our analysis, and describe how to exploit them to evade NIDS. Different from evasion strategies in previous works, we creatively re-use the four-tuple of a connection to exploit some of these discrepancies.

RST rightmost SACK (Discrepancy 7) was introduceds in 2016 [34], as a performance optimization that allows a connection to be closed by a RST promptly. When packet losses or out-of-order packets occur, the rcv_nxt stays at the end of the previously received in-order data. After RFC 5961, TCP only accepts a RST packet if its sequence number is equal to rcv_nxt. So in this case, if an RST is sent after some lost or re-ordered segment, the server's rcv_nxt doesn't match the sequence number in the RST and the server will respond with a challenge ACK. In a lossy situation, the challenge ACK may be lost as well, and the connection will stay alive for a while. Therefore, the newer versions accept a RST packet as long as its sequence number matches the right edge of the right-most SACK block previously received. One can exploit this discrepancy via two possibilities: 1) if the server accepts such RST packets and the NIDS rejects them, then we can send such a RST packet to tear down the connection on the server, and then re-use the four-tuple

Table 2: Discrepancies found between different versions of Linux kernels (from v3.0 to v5.10)

No.	Date	Ver.	Condition	Old Behavior	New Behavior
1	12/3/2011	3.3	SYN+FIN packet in LISTEN state	Initiate a connection	Discard
2	7/17/2012	3.6	In-window SYN packet in ESTABLISHED state	Reset the connection	Discard and send CACK
3	7/17/2012	3.6	In-window RST packet and SEQ number ≠ rcv_nxt	Reset the connection	Discard and send CACK
4	12/22/2012	3.8	Packets with too old ACK number	Accept	Discard and send CACK
5	12/26/2012	3.8	Data packet without ACK flag	Accept the payload	Reject the payload
6	6/13/2013	3.11	Initial receive window when receiving SYN packet	14600	29200
7	6/8/2016	4.8	RST packet with SEQ = end of rightmost SACK block (SACK enabled)	Discard and send CACK	Reset the connection
8	1/17/2017	4.11	RST packet with SEQ = rcv_nxt - 1 in closing states	Discard	Enter CLOSE state
9	5/25/2017	4.13	Data packets in window with old ACK in closing states	Enter CLOSE state	Discard and send CACK

Discrepancies 6, 7, 8, 9 are previously unknown and newly discovered by our system.

to build a new connection with a different initial sequence number (ISN), which will not be tracked by the NIDS; 2) if the NIDS accepts such RST packets and the server rejects them, then we can simply craft such a RST packet to tear down the connection on the NIDS.

RST after FIN (Discrepancy 8) is an optimization to handle compatibility issues with Mac OSX [33]. In Mac OSX, when some applications are abruptly terminated, a RST packet is sent after a FIN packet with the same sequence number as the FIN packet. When a Linux server receives the FIN packet, it advances the rcv_nxt by one; this causes the following RST packet to be rejected because of an out-of-window sequence number, and a challenge ACK to be sent. The MAC OSX client may not reply with any further packets, and the connection on the Linux server will be left in a closing state (e.g., CLOSE_WAIT). To prevent connections from staying in closing states in such cases, the newer versions of the Linux kernel also accepts RST packets with a sequence number equal to rcv_nxt - 1, when in a closing state. One can exploit this discrepancy via two possibilities: 1) if the server accepts such RST packets and the NIDS does not, then we can send such a RST to tear down the connection on the server, and then re-use the four-tuple to build a new connection; because the NIDS has not torn down the old connection, it will not be able to track the new connection; 2) if the NIDS accepts such RST packets and the server does not, then we can send such a RST to tear down the connection on the NIDS, and then re-use the four-tuple to send a SYN packet which will create a half-open connection on the NIDS; after that, we send a legitimate RST packet to tear down the connection on the server, and then re-use the four-tuple to create a new connection with a different ISN; because the NIDS already has a half-open connection, it will miss the new connection.

Data in closing states (Discrepancy 9) will reset the connection in older versions of the Linux kernel because of a buggy implementation [14]. In older versions, when in one of the closing states (e.g., CLOSE_WAIT, CLOSING, LAST_ACK), a data packet with stale sequence and acknowledgment numbers but partial-in-window data will cause the connection to be reset. Although the acknowledgment number is checked, the result is not used, and the packet is not discarded immediately but processed further. In newer versions, this bug was fixed, and such data packets will be discarded and trigger a challenge ACK or duplicate ACK. In order to exploit this discrepancy, there are two possibilities: 1) if the server accepts such data packets and the NIDS does not, then we can send such a data packet to reset the connection on the server, and then re-use the four-tuple to build a new connection; assuming the NIDS has not torn down the old connection, the new connection will not be tracked by the NIDS; 2) if the NIDS accepts such data packets and the server does not, then we can send such a data packet to reset the connection on the NIDS, and then send a SYN packet with the same four-tuple to create a new half-open connection on the NIDS; after that, we send a legitimate RST packet to tear down the connection on the server, and then re-use the four-tuple to create a new connection with a different ISN; the new connection will not be tracked by the NIDS.

5.2 Themis Online Evaluations

In order to understand the effectiveness and efficiency of a NIDS empowered with Themis, we conduct two evaluations. Specifically we assess its robustness against evasion strategies and overhead performance in runtime, and compare our results with a state-of-the-art defense (very recent) [43] against such attacks; this recent approach is based on Deep Learning (DL) models and has disclosed its pipeline implementation and dataset [18].

5.2.1 NIDS Implementation. As mentioned earlier, our NIDS should behave in line with real TCP implementations in Linux, instead of over-simplified implementations as in today's NIDS. However, for ease of implementation, we chose to develop our NIDS on top of Zeek (formerly Bro) [28], one of the most popular open-source general-purpose NIDS in the market. First, we have to realign its behaviors to the common behaviors of the Linux versions we support; Second, we implement the different behaviors regarding each of the discrepancies we discovered; Third, we implement the logics of connection state forking and ambiguity detection.

Overall, we extend Zeek version 4.0 with only 1970 lines of C++ code to handle 8 of the discovered discrepancies listed in Table 2 [1]. Note that realigning Zeek to a Linux implementation is relatively straightforward and introduces negligible overhead.Hereon, we refer to the realigned version of Zeek as ambiguity-agonistic which is the baseline in our overhead evaluation, to be distinguished from Themis which is ambiguity-aware. We open source our implementation and associated datasets on Github[2] for reproducibility and future extensions.

[1] Discrepancy #6 is excluded because the ambiguity can be easily eliminated by looking at the advertised window size in the server's response packet.
[2] https://github.com/seclab-ucr/Themis.

Table 3: Breakdown of ambiguities present in the 8-day MAWI dataset used in evaluations (#6 is excluded as discussed in §5.2.1)

Ambiguity No. (from Table 2)	Total	No Ambiguity	1	2	3	4	5	7	8	9
Count (connections)	72,453,189	72,383,094	5	0	31,149	4,723	3	20	34,343	0
Ratio	100%	99.903%	0.000007%	0%	0.043%	0.0065%	0.000004%	0.00002%	0.047%	0%

5.2.2 Effectiveness. First, we evalute the effectiveness of Themis in defending against evasion attacks. Over the past years, there are a number of evasion strategies proposed [4, 22, 25, 30, 40, 41]. To be comprehensive, we thoroughly analyze all attacks presented in these works and picked strategies that are related to TCP. We summarize and implement 34 different strategies after merging redundant ones; these also include the new strategies discovered by Themis. A detailed list of all implemented strategies can be found in the Appendix. We even design composite strategies that leverage multiple discrepancies in a single connection. Note these strategies fully cover the evaluated attacks in [43], and thus, we are able to conduct an apples-to-apples comparison. Our robust NIDS can detect these attacks with a success rate of 100% (i.e., malicious payloads that are veiled by evasion attacks, can elude the ambiguity-agnostic Zeek but not our robust version). In comparison, [43] reports an Area Under the Receiver Operating Characteristic Curve (AUC-ROC) of 0.963 in detecting state-of-the-art NIDS evasion attacks, meaning it still produces a considerably large number of false positives as well as negatives.

5.2.3 Operational Runtime Overhead. In addition, we evaluate the overheads incurred due to Themis at runtime, compared to the ambiguity-agnostic Zeek when no malicious evasion attacks are present. Note that even without malicious evasion attacks, there can be a number of ambiguous packets observable in natural network traffic. This is because in a wild Internet environment, various implementations may exist and the packets exchanged across them may satisfy the conditions associated with ambiguities. In such cases, these ambiguities are not actively exploited for malicious purposes, but can still cause overhead since Themis would still fork states on such bases. We refer to this overhead as operational runtime overhead associated with Themis when it is deployed in real network environments.

The key to accurately estimating the operational overhead is finding representative network traffic captures to evaluate Themis. For this, we use the MAWI Traffic Archive [17] as the base dataset. It provides PCAP dumps from a backbone network located in Japan, and is thus, considered sufficiently large and representative. We pick 7-day recent traces captured from April 25 to May 1, 2021, in additon to the trace on April 7, 2020 (i.e., the dataset used in [43]), and filter out any non-TCP connections to forge a test set of 72,453,189 TCP connections. Table 3 shows the statistics of different ambiguities present in the trace. Overall, only a very small fraction of natural / benign traffic contain packets that cause ambiguities. Specifically, there are only 69,994 (0.097% of the connections) connections with exactly 1 ambiguity, 131 with 2 different ambiguities, 1 with 3 different ambiguities, and no connections with more than 3 ambiguities. We use the same machine with an AMD EPYC 7542 Processor as in §5.1, but only use a single core in order to be comparable with

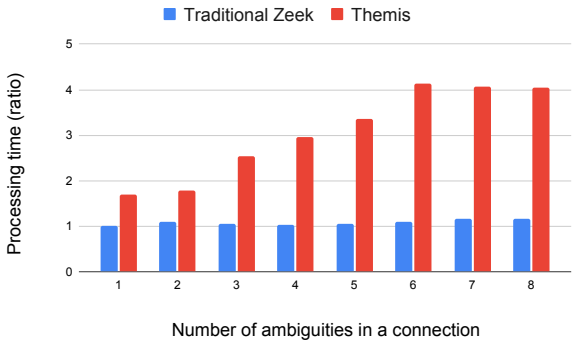

Figure 8: Overhead growth with the number of ambiguities

[43]. We record the average memory usage to be 1~2 GB, which is on par with (if not less than) that in [43]. As for the resulting operational overhead, we find that compared to ambiguity-agnostic Zeek, our robust version incurs only about 1.07% additional processing time, indicating only negligible levels of operational cost. The average processing time is 69400.5 packets per second. In comparison, the state-of-the-art defense from [43], can only process less than 2200 packets per second (due to the computational cost of the deep learning model), which is more than 30 times slower than Themis.

5.2.4 Overhead Growth with Multiple Ambiguities. In addition to the operational overhead incurred on benign traffic traces, we also evaluate the runtime overhead that Themis imposes in the presence of multiple different ambiguities in a single connection. Although, natural traffic rarely includes more than one ambiguity in the same connection (only ~0.0002% in our 8-day MAWI dataset), we are interested in knowing how would the overhead of Themis grow with repsect to the number of ambiguities, for understanding how vunlunerable Themis is agaisnt Denial-of-Service attacks (i.e., deliberately injected multiple ambiguities for slowing down NIDS processing). To maliciously induce extremely high overhead, an attacker could aim to trigger as many state forkings as possible for each connection. Because each packet will be processed by all forked states, the overhead is proportional to the number of forked states. Furthermore, the attacker may want to use long-lived connections, because after reaching the maximum number of forked states, every packet sent by the attacker will cause significant extra overhead on the NIDS. Based on this reasoning, we manually inject ambiguities into normal connections, and measure the processing time growth of Themis along with number of ambiguities. The results are shown in Figure 8, and suggest that the overhead growth is in principle linearly proportional to the number of ambiguities. Note that the overhead stops growing after 6 ambiguities. This is because some ambiguities, specifically 7, 8, and 9, cannot coexist in the same connection due to version coherence and connection

being reset. For example, versions before ambiguity 7 all conform with the old behavior for ambiguity 7, 8 and 9, and for versions after ambiguity 7, the connection will be reset after seeing ambiguity 7. In addition, based on the results in §5.2.3, it's rare to encounter a connection with more than 3 ambiguities, therefore, we could safely mark a connection as suspicious if it has more than 3 ambiguities.

6 DISCUSSION AND LIMITATIONS

Completeness and Soundness of Our Symbolic Modeling. By performing an exhaustive symbolic execution, we achieve a complete coverage of feasible execution paths by finishing all execution states. However, there still could be potential cases missing. First, we use the default configuration of Linux (e.g, kernel compilation configuration, Linux sysctl settings), and thus, cannot guarantee all TCP logic is covered (e.g, some uncommon features like TCP Fast Open may not be enabled by default). Second, we eliminate some non-determinism in TCP (e.g., we freeze the CPU clock), and this could also cause incomplete coverage because we only explore one possibility instead of all. Further, some TCP logics need to be triggered by user space applications, which can set TCP socket options, or call certain system calls like `connect()`, `send()`, `recv()`, etc. In our experiments, we use a simple server-side application to passively listen on a socket and receive packets, since our target is to infiltrate data into a server's receive buffer. We use the default TCP socket options, which are also used by other network applications, such as the Apache HTTP server. Our model can already cover the major logic of TCP, and we believe that our setting is representative but acknowledge that it still may not be complete.

In addition, we note that there is a data overlapping strategy [40] missed by Themis. Basically, an attacker can craft two data packets with overlapping portions in payloads and result in ambiguities regarding which copy will be accepted. Most of the OSes, such as Linux, favor the last packet, while Windows favors the first packet. Since Themis currently analyzes only Linux implementations, it cannot discover this particular discrepancy. Nevertheless, it can still be easily incorporated into the ambiguity-aware NIDS.

Extent of Human Effort in Symbolic Model Extraction and Comparison. When labeling merge ranges for state merging, we first automatically generate a list of candidate merge ranges with symbolic execution and control flow analysis; we then go over them manually to decide whether to label them as merge ranges or not, guided by our TCP domain expertise and using certain heuristics. In addition, we also need to label the program points related to critical states. It takes on average, one day for someone with domain expertise to do the above labeling for a Linux kernel version. After symbolic model extraction, we employ a constraint solver to automatically compare two models and find concrete examples for any discrepancies, which does not require any human effort. Once a discrepancy is found, we need to manually analyze the execution traces and summarize the discrepancy. Therefore, the number of discrepancies will dictate the manual efforts for discrepancy analysis. In the evaluation of the offline phase, it takes us around 10 days in total to finish analyzing 5 Linux kernel versions and find all the discrepancies across these versions.

Assumptions on Version Coherence. As discussed in §4.2, version coherence is an optimization to reduce the number of forked states in an NFA-based NIDS. Its aim is to build a profile for each version range in which all the versions have the same TCP behaviors. It is an optional feature because as discussed in §5.2.3, it is very rare that a connection has more than 3 ambiguities; thus, we can safely mark a connection as suspicious when we see such a connection, without doing further state forking. When building profiles for the Linux TCP stack, we choose to compare adjacent versions rather than all pairs of versions. The underlying assumption is that we work on a single implementation with an ordered version history, i.e., we can know which version precedes another. When comparing versions from different OSes, we will have to do a full pair-wise comparison between all the versions.

Extending to Other Operating Systems. Although Linux enjoys a major market share among the server OSes, there are also other OSes such as Windows and FreeBSD. Themis uses S2E [13] as the symbolic execution engine to extract the TCP model from an OS. S2E works on the binary level and runs the entire OS in QEMU, and does not require source code. So in principle, we could extend Themis to all other OSes, including those that are closed-source (e.g., Windows). However, we will need to label the critical states and merge ranges to scale up symbolic execution. Without access to source code, this process can be more time-consuming.

Extensions to Model Client Behaviors. In this work, we focus on modeling servers' behaviors. Although the TCP stack of the client and the server are the same, we do not explore TCP states exclusively related to the client, e.g., TCP_SYN_SENT. Our motivation stems from the fact that NIDS are typically deployed as safeguards against servers in corporations as opposed to individual clients which could be anywhere in the world. In order to model the clients' behaviors, we need to run a client application and actively initiate connections and send packets. Conceivably, Themis can protect a client from being exploited by malicious content sent from a server. However, we leave this possibility to future work.

Ethical Considerations. We acknowledge that improving the robustness of NIDS has an unintended consequence of improving the robustness of censorship firewalls too, as they both need to keep track of TCP connection states and reassemble TCP data packets. Similar to other technologies such as encryption that can be used for both good (e.g., protecting our privacy) and bad purposes (e.g., plotting a terrorist attack), we believe the value in preventing malicious attacks generally outweighs the collateral damage of disrupting censorship circumvention.

Limitations. Since Themis-enabled NIDS is designed to address TCP-layer ambiguities, it may fall short when attackers leverage ambiguities from other layers, e.g., the IP-layer TTL trick (constructing packets with smaller TTL values that can reach the NIDS but not the remote server). In this case, one could adopt a defense-in-depth strategy and deploy a traffic normalizer [19], which can defend against the TTL trick by increasing the TTL value in a packet.

7 RELATED WORK

Finding Discrepancies between Implementations. Discrepancies between implementations are usually good indicators of implementation bugs. They can also be used to fingerprint implementations or evade detection (as considered here) leveraging semantic

gaps [21, 41]. There is work on finding discrepancies between different implementations of the same target; examples include network protocols [7, 12, 41], parsers [9, 21, 29], libraries [38, 39], etc. A common way to find discrepancies is differential testing combined with random input generation or fuzzing. Brubaker et al.[6], generate synthetic X.509 certificates by randomly mutating fields in a real certification, and then feed them to different certificate validation programs in order to find bugs from discrepancies. Jana et al. [21] also employ differential fuzzing but against malware detectors. They discover novel attacks that exploit the discrepancies between parsers of the malware detectors and actual applications, and can evade the detection. This approach treats the target as a black-box and does not require any internal information, and is therefore easy to apply. However, the coverage is usually low because it can only explore the search space near the seed input.

Some other works use static analysis to extract semantic information from binary or source code. Min et al. [27], target the Linux file systems and extract high-level semantic information from the source code; they then do a statistical comparison to discover deviant behaviors. Srivastava et al. [38] conduct flow- and context-sensitive interprocedual static analysis on Java API implementations, and produce context-sensitive security policies for every API entry point, and then compare the policies to find discrepancies. Static analysis can leverage semantic information and is scalable, but also suffers from false positives.

Symbolic execution is also used to extract a more accurate semantic representation from the source code or binary. Brumley et al. [7], extract symbolic formulas by replaying captured network traces against different implementations, and then compare the symbolic formulas with a constraint solver. But due to limited coverage, they suffer from false positives. Similarly, Chau et al [11] feed symbolic X.509 certificates to certificate validation implementations, and extract constraints relating to certificate "accept" and "reject" paths. They then use a constraint solver to find discrepancies. Wang et al. [41] combine symbolic execution with black-box differential testing, and use symbolic execution as a guide to group equivalence inputs by execution paths and therefore, largely reduce the search space. However, all these works only achieve partial coverage and compare indiviual execution paths to discover discrepancies opportunistically. Our approach aims to systematically discover all discrepancies between two implementations, as it relates to NIDS evasion. To achieve this goal, we need to run symbolic execution exhaustively to traverse all feasible execution paths in each of our target implementations, and then calculate the weakest preconditions based on the entire program.

Defenses against NIDS Evasion Attacks. Zhu et al.[43] present a deep learning based solution for detecting and localizing DPI evasion attacks by learning the so-called packet context (i.e., interrelationships of header fields within and across packets) from benign traffic traces. It then uses the learnt model on unseen network connections to spot anomalies in terms of deviations from the benign context distribution. As discussed in §5, compared to Themis, [43] falls behind in terms of both the detection accuracy (0.963 vs. 1.00 in AUC-ROC) and runtime overhead (2162.2 vs. 69400.5 packets processed per second under the same single-core CPU setup). This is because any DL-based defense, unlike Themis, always will generate some incorrect classifications and require relatively heavy computations in their inference phase.

Traffic normalization [19] takes a different approach in defending NIDS against evasion attacks. A normalizer sits in the path and patches up the packets passing through to eliminate potential ambiguities, before they are seen by the NIDS. It relies on a manually curated list of potential ambiguities in basic network protocols such as TCP, UDP, IP, and ICMP. However, unfortunately, it cannot safely remove all possible ambiguities in the absence of detailed knowledge about the various implementations on the endhosts, and could even disrupt the communications since it alters the traffic.

Active Mapping [35] builds a profile for each endhost and actively maintains a profile database. This is apt for a small network with relatively stable members, and not for a large scale network with dynamic members. Usually, it takes time to build profiles, and they need to be updated often. Moreover, the NIDS needs to be agnostic to the details and configurations of software on the endhosts.

Paxson [28] proposes to use bifurcating analysis to explore all different possibilities of packet reassembly. However, without the knowledge of ambiguities, it will lead to exponential growth in state forking overhead in practice.

8 CONCLUSIONS

In this paper, we aim to defend against attacks that seek to evade network intrusion detection systems, by exploiting the discrepancies between its TCP implementation and that at a targeted end server. These discrepancies are commonplace, and, thus these threats are very real. We design a novel lightweight system Themis which is extremely effective in defending against such attacks. It contains an offline phase, where it identifies and models discrepancies in TCP implementations across OS versions using symbolic execution. The models are then employed at runtime, and by applying a non-deterministic automaton the proper implementation versions are forked to handle packets correctly and block evasion attempts. Themis is extremely effective and is able to block all known evasion attempts to date with negligible additional overhead on a NIDS. In developing Themis we also discover multiple brand new discrepancies, that are exploitable as it relates to current NIDS.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers and our shepherd, Soo-Jin Moon, for their insightful feedback that helped improve the quality of this paper. This research was partially sponsored by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on. This research was also partially sponsored by NSF grant No. 1652954.

REFERENCES

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages. https://doi.org/10.1145/3182657

[2] Kevin Bock, Yair Fax, Kyle Reese, Jasraj Singh, and Dave Levin. 2020. Detecting and Evading Censorship-in-Depth: A Case Study of Iran's Protocol Whitelister. In *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*. USENIX Association. https://www.usenix.org/conference/foci20/presentation/bock

[3] Kevin Bock, George Hughey, Louis-Henri Merino, Tania Arya, Daniel Liscinsky, Regina Pogosian, and Dave Levin. 2020. Come as You Are: Helping Unmodified Clients Bypass Censorship with Server-Side Evasion. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 586–598. https://doi.org/10.1145/3387514.3405889

[4] Kevin Bock, George Hughey, Xiao Qiang, and Dave Levin. 2019. Geneva: Evolving Censorship Evasion Strategies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2199–2214. https://doi.org/10.1145/3319535.3363189

[5] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6 (1975), 234–245.

[6] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 114–129.

[7] David Brumley, Juan Caballero, Zhenkai Liang, and James Newsome. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *16th USENIX Security Symposium (USENIX Security 07)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/16th-usenix-security-symposium/towards-automatic-discovery-deviations-binary

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[9] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors.. In *NDSS*.

[10] Sze Yiu Chau, Omar Chowdhury, Md. Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 503–520. https://doi.org/10.1109/SP.2017.40

[11] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS# 1 v1. 5 Signature Verification.. In *NDSS*.

[12] Jianjun Chen, Jian Jiang, Haixin Duan, Nicholas Weaver, Tao Wan, and Vern Paxson. 2016. Host of Troubles: Multiple Host Ambiguities in HTTP Implementations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1516–1527. https://doi.org/10.1145/2976749.2978394

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[14] Data in closing states [n.d.]. *tcp: better validation of received ack sequences*. Retrieved April 17, 2021 from https://github.com/torvalds/linux/commit/d0e1a1b5a833b625c93d3d49847609350ebd79db

[15] Edsger W. Dijkstra. 1976. *A Discipline of Programming*. Prentice-Hall. I–XVII, 1–217 pages.

[16] Reham Taher El-Maghraby, Nada Mostafa Abd Elazim, and Ayman M Bahaa-Eldin. 2017. A survey on deep packet inspection. In *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. IEEE, 188–197.

[17] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. 2010. Mawilab: combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *Proceedings of the 6th International COnference*. 1–12.

[18] Inc. GitHub. 2020. Source Code and Dataset for CLAP. https://github.com/seclab-ucr/CLAP.

[19] Mark Handley, Vern Paxson, and Christian Kreibich. 2001. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-end Protocol Semantics. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (Washington, D.C.) *(SSYM'01)*. USENIX Association, Berkeley, CA, USA, 9–9. http://dl.acm.org/citation.cfm?id=1267612.1267621

[20] Trevor Hansen, Peter Schachte, and Harald Søndergaard. 2009. *State Joining and Splitting for the Symbolic Execution of Binaries*. Springer-Verlag, Berlin, Heidelberg, 76–92. https://doi.org/10.1007/978-3-642-04694-0_6

[21] Suman Jana and Vitaly Shmatikov. 2012. Abusing File Processing in Malware Detectors for Fun and Profit. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, USA, 80–94. https://doi.org/10.1109/SP.2012.15

[22] Sheharbano Khattak, Mobin Javed, Philip D. Anderson, and Vern Paxson. 2013. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *Presented as part of the 3rd USENIX Workshop on Free and Open Communications on the Internet*. USENIX, Washington, D.C. https://www.usenix.org/conference/foci13/workshop-program/presentation/Khattak

[23] James C. King. 1975. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). Association for Computing Machinery, New York, NY, USA, 228–233. https://doi.org/10.1145/800027.808444

[24] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient State Merging in Symbolic Execution. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 193–204. https://doi.org/10.1145/2254064.2254088

[25] Fangfan Li, Abbas Razaghpanah, Arash Molavi Kakhki, Arian Akhavan Niaki, David Choffnes, Phillipa Gill, and Alan Mislove. 2017. Lib·erate, (N): A Library for Exposing (Traffic-classification) Rules and Avoiding Them Efficiently. In *Proceedings of the 2017 Internet Measurement Conference* (London, United Kingdom) *(IMC '17)*. ACM, New York, NY, USA, 128–141. https://doi.org/10.1145/3131365.3131376

[26] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.

[27] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-Checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 361–377. https://doi.org/10.1145/2815400.2815422

[28] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-time. *Comput. Netw.* 31, 23-24 (Dec. 1999), 2435–2463. https://doi.org/10.1016/S1389-1286(99)00112-7

[29] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 615–632. https://doi.org/10.1109/SP.2017.27

[30] Thomas H Ptacek and Timothy N Newsham. 1998. *Insertion, evasion, and denial of service: Eluding network intrusion detection*. Technical Report. SECURE NETWORKS INC CALGARY ALBERTA.

[31] Anantha Ramaiah, R Stewart, and Mitesh Dalal. 2010. *Improving TCP's Robustness to Blind In-Window Attacks*. RFC 5961. RFC Editor. 1–19 pages. https://www.rfc-editor.org/rfc/rfc5961.txt

[32] Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 332–344. https://doi.org/10.1145/3427228.3427662

[33] RST after FIN [n.d.]. *tcp: accept RST for rcv_nxt - 1 after receiving a FIN*. Retrieved April 17, 2021 from https://github.com/torvalds/linux/commit/0e40f4c9593ba2c7c30150ed669da97bd581c0cd

[34] RST rightmost SACK [n.d.]. *tcp: accept RST if SEQ matches right edge of right-most SACK block*. Retrieved April 17, 2021 from https://github.com/torvalds/linux/commit/e00431bc93bb48c650273be4a00007b2a392d32a

[35] U. Shankar and V. Paxson. 2003. Active mapping: resisting NIDS evasion without altering traffic. In *2003 Symposium on Security and Privacy, 2003.* 44–61. https://doi.org/10.1109/SECPRI.2003.1199327

[36] Michael Sipser. 1996. *Introduction to the Theory of Computation* (1st ed.). International Thomson Publishing, 47–48.

[37] SMT-LIB [n.d.]. *SMT-LIB The Satisfiability Modulo Theories Library*. Retrieved April 20, 2021 from http://smtlib.cs.uiowa.edu

[38] Varun Srivastava, Michael D Bond, Kathryn S McKinley, and Vitaly Shmatikov. 2011. A security policy oracle: Detecting security holes using multiple API implementations. *ACM SIGPLAN Notices* 46, 6 (2011), 343–354.

[39] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. 2020. Discovering Discrepancies in Numerical Libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 488–501. https://doi.org/10.1145/3395363.3397380

[40] Zhongjie Wang, Yue Cao, Zhiyun Qian, Chengyu Song, and Srikanth V. Krishnamurthy. 2017. Your State is Not Mine: A Closer Look at Evading Stateful Internet Censorship. In *Proceedings of the 2017 Internet Measurement Conference* (London, United Kingdom) *(IMC '17)*. ACM, New York, NY, USA, 114–127. https://doi.org/10.1145/3131365.3131374

[41] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V Krishnamurthy, Kevin S Chan, and Tracy D Braun. 2020. SymTCP: eluding stateful deep packet inspection with automated discrepancy discovery. In *Network and Distributed System Security Symposium (NDSS)*.

[42] Z3Prover/z3 [n.d.]. *The Z3 Theorem Prover*. Retrieved May 4, 2019 from https://github.com/Z3Prover/z3

[43] Shitong Zhu, Shasha Li, Zhongjie Wang, Xun Chen, Zhiyun Qian, Srikanth V. Krishnamurthy, Kevin S. Chan, and Ananthram Swami. 2020. You Do (Not) Belong Here: Detecting DPI Evasion Attacks with Context Learning. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 183–197. https://doi.org/10.1145/3386367.3431311

A LIST OF IMPLEMENTED EVASION STRATEGIES FOR EFFECTIVENESS EVALUATION OF THEMIS

We have implemented all the TCP-related evasion strategies presented in previous works [4, 22, 25, 30, 40, 41], after merging redundant strategies. There are in total 34 evasion stategies as listed in Table 4. We apply those strategies to a HTTP connection, in which the client sends a malicious request with a bad keyword. Themis can successfully detect the bad keyword in all the attacks.

Table 4: List of implemented evasion strategies (bold strategies are newly discovered in this paper)

No.	Strategy	Description
1	Bad checksum data	In ESTABLISHED state, send junk data with bad checksum, and then send the request
2	Bad checksum RST	In ESTABLISHED state, send partial request, then send a RST packet with bad checksum, and then send the remaining request
3	No ACK flag data	In ESTABLISHED state, send junk data without ACK flag, and then send the request
4	No ACK flag FIN	In ESTABLISHED state, send partial request, then send a FIN packet without ACK flag, and then send the remaining request
5	SYN with data	In LISTEN state, send a SYN packet with payload, then send the request
6	Bad ACK number data	In ESTABLISHED state, send junk data with out-of-window ACK number, and then send the request
7	Bad ACK number RST/ACK	In ESTABLISHED state, send partial request, then send a RST/ACK packet with out-of-window ACK number, and then send the remaining request
8	Small data offset header	In ESTABLISHED state, send junk data with TCP data offset <5, and then send the request
9	Large data offset header	In ESTABLISHED state, send junk data with TCP data offset >actual packet size / 4, and then send the request
10	Bad MD5 data	In ESTABLISHED state, send junk data with TCP MD5 option, and then send the request
11	Bad MD5 RST	In ESTABLISHED state, send partial request, then send a RST packet with TCP MD5 option, and then send the remaining request
12	Bad TCP timestamp data	In ESTABLISHED state, send junk data with bad TCP timestamp, and then send the request
13	Bad TCP timestamp RST	In ESTABLISHED state, send partial request, then send a RST packet with bad TCP timestamp, and then send the remaining request
14	Bad SEQ number data	In ESTABLISHED state, send junk data with out-of-window SEQ number, and then send the request
15	Bad SEQ number FIN	In ESTABLISHED state, send partial request, then send a FIN packet with out-of-window SEQ number, and then send the remaining request
16	Bad SEQ number RST	In ESTABLISHED state, send partial request, then send a RST packet with out-of-window SEQ number, and then send the remaining request
17	Invalid TCP flags	In ESTABLISHED state, send junk data with flags FRAPUN set, and then send the request
18	Multiple SYNs	In SYN_RECV or ESTABLISHED state, send a SYN packet with out-of-window SEQ num, and then send the request
19	Big gap in data	In ESTABLISHED state, send junk data with SEQ = rcv_nxt + max_gap_size (16384), and then send the request
20	SEQ number before ISN	In ESTABLISHED state, send the request with SEQ <ISN (initial sequence number) but partial-in-window data
21	In-window SYN	In ESTABLISHED state, send partial request, then send a SYN packet with SEQ >rcv_nxt but in window, and then send the remaining request
22	In-window FIN	In ESTABLISHED state, send partial request, then send a FIN packet with SEQ >rcv_nxt but in window, and then send the remaining request
23	In-window RST	In ESTABLISHED state, send partial request, then send a RST packet with SEQ >rcv_nxt but in window, and then send the remaining request
24	Partial in-window RST	In ESTABLISHED state, send partial request, then send a RST packet with SEQ <rcv_nxt but partial data in window, and then send the remaining request
25	Urgent data	In ESTABLISHED state, send the request with urgent pointer and URG flag set, also need to insert one byte urgent data into the payload
26	Time gap	In ESTABLISHED state, send partial request with timestamp, and then send the remaining request with timestamp = last_timestamp + 0x80000000
27	Small segments	In ESTABLISHED state, send the request in small segments (size = 4)
28	TCB Turnaround	In LISTEN state, send a SYN/ACK packet before sending the SYN packet, then establish the connection and send the request
29	Muti-segmentation	In ESTABLISHED state, send the request in segments, 1st segment size 8, 2nd segment size 4, and then send the remaining request
30	Simple TCB Desynchronization	In SYN_RECV state, send a SYN packet with junk payload, and then send the request
31	SYN+FIN	In LISTEN state, send a SYN+FIN packet, then establish the connection with a different ISN and send the request
32	**RST rightmost SACK**	SAckOK option in SYN packet. In ESTABLISHED state, send partial request with a SEQ gap, then send a RST packet with SEQ = SEQ end of last packet, and then send the remaining request
33	**RST after FIN**	In ESTABLISHED state, send a FIN/ACK packet, then send a RST packet with SEQ = SEQ of the FIN packet, and then reuse the 4-tuple to established a new connection and send the request
34	**Data in closing states**	In ESTABLISHED state, send a FIN/ACK packet, then send junk data with SEQ <rcv_nxt but in-window data and ACK <previous ACK, and then reuse the 4-tuple to establish a new connection and send the request