PathMiner Powered Predictable Packet Processing

John Sonchack and Jonathan M. Smith
University of Pennsylvania
{ jsonch, jms }@cis.upenn.edu

Abstract—**Performance advances have made software packet processing a compelling alternative to hardware. However, software still lacks the *delay predictability* of hardware, an important property for security and quality of service. As a solution, we introduce `PathMiner`, an analysis tool that automatically builds performance models for software packet processors at the fine granularity of per-packet execution times. `PathMiner` combines symbolic execution and genetic algorithms in an iterative feedback loop to rapidly mine a packet processor binary for diverse packets that invoke complex execution paths. With these packets, `PathMiner` trains machine learning models that predict packet execution times and paths based on raw packet header bytes. We implement a prototype of `PathMiner` and test it by profiling a software `IP` router. The evaluation shows that `PathMiner`'s models predict delay with low error – for over 40% of packets they predicted the router's execution time to within 10 cycles. Closer examination shows that `PathMiner`'s effectiveness is due to higher execution path coverage than symbolic execution and the capability to generate diverse training samples efficiently. `PathMiner` takes an important step towards making it practical to predict delay for any software packet processor.**

I. Introduction

Over the past decade, the popularity of software packet processing has grown alongside the desire for flexible and programmable networks [10]. While the traditional downside of software has been performance, closing the gap with hardware has been an important ongoing effort for the networking community [22], [1], [28].

As a result, overall performance of software packet processors has increased to the point where they can meet the throughput needs of high speed networks [8]. However, there is another important but less addressed gap between hardware and software, *delay predictability*: ***How long will a network element spend processing a packet?*** Delay predictability, especially at the tail, is critical for many applications. For example, soft real-time systems [26], [19] need estimates of worst case delays for scheduling; DoS defense systems [4] need to identify execution paths that saturate compute resources of network elements; and data center networks need to ensure low tail latency [20] to optimize user experience.

With hardware packet processors it is simple to predict delay because their data paths are designed to meet strict timing deadlines, e.g., processing one packet per cycle [29], [2], [15] to guarantee line rate. Unfortunately, there are no such guarantees for software packet processors, not least of all due to their complexity. Software is used for more advanced functionality than hardware, *e.g.*, routing instead of switching, which will cause unpredictability even if we model commodity servers perfectly and use real time operating systems [36]. Software packet processors follow the well-known rule [7] of having exponentially many unique execution paths with respect to program length. Latency varies greatly across execution path, the selection of which can depend on a packet in complex ways. For example, two packets that differ by only a single bit in their `IP` headers can invoke significantly different execution paths in a router if one packet has a valid `IP` header checksum and the other does not.

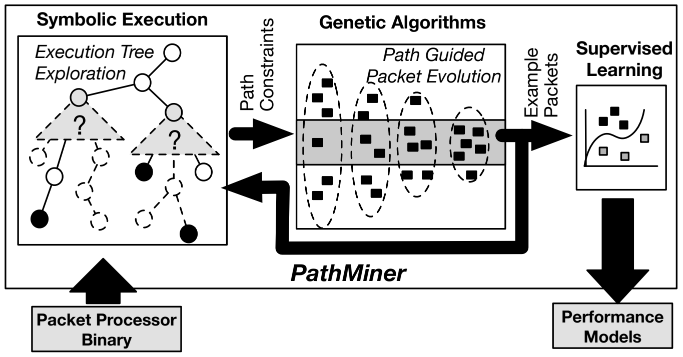

Fig. 1. `PathMiner` mines packet processors for *execution paths* and *sample packets* to train models.

All of this complexity makes it challenging to predict delays, especially at the tail, for software packet processors. Empirical profiling with random or pre-recorded packet traces can estimate delays for some of the common execution paths, but is unlikely to trigger those responsible for high delays. Formal analysis techniques, such as symbolic execution, can systematically search for longer execution paths but struggle with complex control flows, e.g., that have many branch points. Current formal approaches require manual effort to overcome the challenge, such as code rewriting [37] and annotation [7], or only work for applications built in specialized frameworks that are amenable to analysis [32], [3]. This limits practicality because operators are unlikely to modify source code, and worse, commercial software is often distributed as binaries.

Introducing PathMiner. In this paper, we take a step towards making delay predictability practical for *any* software packet processor. We introduce `PathMiner`, a fully automated system that analyzes a packet processor binary to build models of it at the fine granularity of per-packet execution. The models

Workshop on Binary Analysis Research (BAR) 2018
18 February 2018, San Diego, CA, USA
ISBN 1-1891562-50-9
https://dx.doi.org/10.14722/bar.2018.23020
www.ndss-symposium.org

predict execution paths and times using raw packet header bytes as input features. They are potentially useful for many applications. For example, a real time system [26] could use the models for a more accurate estimate of tail latency in networks with software packet processors, while a DoS defense system [4] could use the models to identify execution paths that consume many cycles and proactively rate limit packets that are predicted to invoke those paths.

To build accurate models, PathMiner must find complex execution paths in the binary and generate diverse training samples that represent the range of possible packets that may invoke those paths. PathMiner achieves these goals by setting up a tight feedback loop between *symbolic execution* (SE) and *genetic algorithms* (GAs) as shown in Figure 1. PathMiner uses SE to search for feasible execution paths with the input packet represented symbolically. Whenever SE reaches a branch in the control flow that is too complex for symbolic logic to solve efficiently, PathMiner turns to GAs. Using the constraint formula, PathMiner derives parameters for the GA that pressures it to generate *packets that reach the branch and then follow long execution paths.* These packets are fed back to the SE as seeds that provide it with a skeleton of long paths beyond the branch, so it can make progress without having to solve the complex symbolic formula. Whenever SE identifies a complete execution path through the binary, PathMiner uses the same GA technique to generate randomized packets that invoke it; these packets are labeled with execution time and path ID and used to train models.

Prototype Implementation and Results. We implemented a prototype of PathMiner using S2E [5], Pyevolve [25], and Scikit [24]. The prototype includes a harness to analyze the binaries of Click [22] packet processors.

We profiled a full IP router [22] for 12 hours using the prototype and found that:

- PathMiner's models predicted delay accurately based on raw packet header bytes. They predicted router execution times to within 10 cycles for over 40% of packets and had a median error of 0.
- PathMiner provided significantly higher coverage than S2E by itself. It found over $4X$ more execution paths and 50% higher execution times in the analysis.
- PathMiner's models could also go beyond latency, predicting which execution path the router would use to process a packet with high accuracy and recall, .96 and .92.

II. Background

A general tool for building ML models that predict packet delays and execution paths of unmodified packet processor binaries is clearly desirable. The core challenge is sampling a diverse set of execution paths in the binary and efficiently generating training packets that represent them. PathMiner leverages two search-based binary analysis techniques to meet this goal: *symbolic execution* and *genetic algorithms*. As we describe in this section, each technique is powerful but has limitations for discovering execution paths and generating training packets.

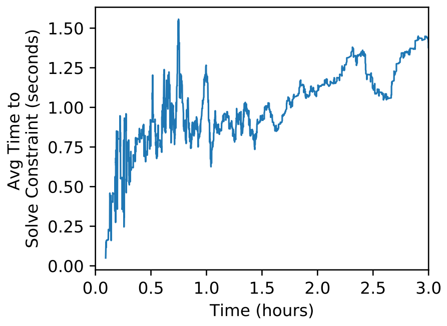

Fig. 2. Average time spent in constraint solver per branch instruction, while profiling the IP router with S2E.

A. Symbolic Execution (SE)

The key idea behind symbolic execution [21] is to execute a program with symbolic inputs that can represent any values, which allows an analysis engine to identify execution paths that the program would take under different input values. The program runs in a symbolic execution engine that acts as an interpreter and represents program variables as formulas over the symbolic values. As execution proceeds, the engine builds an *execution tree* that represents feasible execution paths, i.e., those possible with some concrete input values. Each node in the tree corresponds to executing a basic block of code and stores a first-order *path constraint formula* that represents all of the constraints the symbolic inputs must have met to reach that point in the execution. Each edge represents a transition between basic blocks. If the exit point of a basic block is a conditional branch with operands that depend on the symbolic input, the symbolic engine uses a constraint solver to determine which branches are feasible based on the branch condition and the constraint formula of the parent node. The symbolic engine adds a child node for every feasible branch, with an updated path constraint formula to account for the branch condition.

Execution Path Discovery. Symbolic execution can find short execution paths quickly. However, its exploration rate quickly asymptotes when it reaches longer paths. This biases its overall discovery towards short paths. For example, in a 3 hour analysis of the IP router with the S2E SE engine, the discovery rate drops from multiple paths per second to under 1 new path per hour. There are at least two underlying causes: complex path constraints and path explosion.

First, as the SE engine reaches longer execution paths, exploration slows because it spends more time in the constraint solver. The number of the path constraints grows linearly with execution path length, and the cost of solving a constraint formula also grows at least linearly [14]. As Figure 2 shows, after 2 hours of profiling an IP router with S2E, its constraint solver (z3 [6], one of the most efficient solvers) spent multiple seconds, on average, solving the constraint for *each branch*.

Second, the number of feasible paths through a program grows exponentially with program size, and is often infinite, e.g., due to unbounded loops [21]. This *path explosion* is a well-known issue for SE and makes it difficult to find a representative sample of long execution paths. The SE engine does not have the time to evaluate every possible path, and does not know which selection of branches will lead to the long execution paths that the profiler seeks.

Training Sample Generation. The SMT solver generates a concrete input that invokes each complete execution path, in order to prove that the path is feasible. This provides one training sample per path, but generating additional samples with the SMT solver is expensive. It requires re-evaluating the formula with additional constraints that explicitly disallow the previous solutions. This is not efficient when a large number of samples are required, e.g., for machine learning. Worst, successive samples are often very similar to each other, as solvers tend to search for satisfying assignments with as few changes as possible, for performance.

B. Genetic Algorithms

Genetic Algorithms (GAs) [17] are a stochastic search technique inspired by evolutionary *survival of the fittest* principles. In a GA, solutions to a problem are encoded into a *chromosome*. A *fitness function* maps a chromosome to a score that measure its quality as a solution. A GA begins with a *population* of randomly generated chromosomes, and iteratively evolves the population to increase fitness. To evolve the population, from a parent generation to a child generation, the GA first selects pairs of fit chromosomes from the parent generation. It *mutates* the selected chromosomes by replacing parts of them with random values, with low probability. Finally, it applies a *crossover* operation to each chromosome pair, which swaps some of their values. The resulting chromosomes are optionally *repaired* with a custom function, to ensure that they represent valid solutions, then placed into the next generation. A GA evolves new generations until the population converges with respect to fitness score.

GAs are appealing because they are unsupervised. The user only needs to provide input on the "shape" of a solution and how to score its quality. Additionally, GAs tend to converge quickly, because the parts of chromosomes that lead to high fitness scores in one generation are exponentially likely to appear in the next [11]. These properties have led GA to be used for NP-hard optimization problems in many fields.

Execution Path Discovery. GAs have been used to estimate worst case execution time in real-time systems [27], [35], with fitness functions that score high execution times positively. This can be an effective way to identify a few long execution paths. However, in the IP router, we found that GAs tended to get stuck in longer than average paths that were easy to invoke with random packets. The GAs converged on these paths and their populations quickly lost diversity, which prevented them from exploring other parts of the execution tree. For example, paths for `ARP` processing, which were invoked for any packet with an Ethernet type of `0x0806`. GAs often converged on the `ARP` paths, which prevented them from making progress towards finding longer execution paths for processing `IP` packets with valid checksums.

Training Sample Generation. Once a GA is locked into an execution path, it can rapidly produce randomized inputs that invoke the path by simply evolving additional generations. This is faster than an SMT solver, and yeilds more diverse samples.

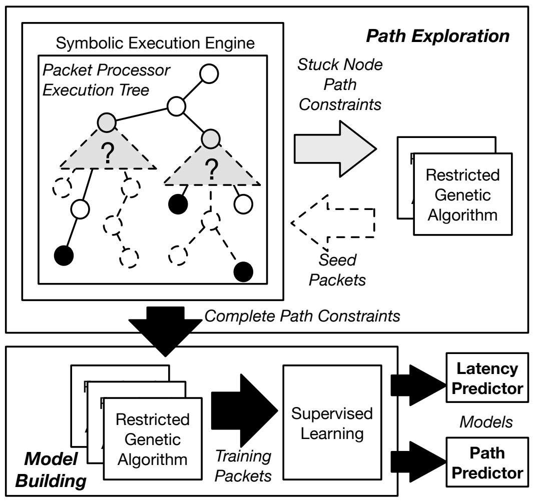
Symbolic Execution Engine
Path Exploration
Packet Processor Execution Tree
?
?
Stuck Node Path Constraints
Restricted Genetic Algorithm
Seed Packets
Complete Path Constraints
Restricted Genetic Algorithm
Training Packets
Supervised Learning
Latency Predictor
Models
Path Predictor
Model Building

Fig. 3. Overview of `PathMiner`.

However, the challenge, which `PathMiner` addresses, is getting the GA to converge on the desired execution path.

III. INTRODUCING PATHMINER

`PathMiner` is based on the observation that GAs and SE have complementary search biases and sample generation properties. SE can methodically search through the execution tree of a packet processor to find representative paths, but has difficulties finding long paths and efficiently generating training examples. On the other hand, GAs can find long paths and generate examples efficiently, but tend to get stuck in local optima.

Figure 3 summarizes `PathMiner`'s architecture. The input is a packet processor binary and the outputs are classifiers that predict execution times and paths using the first 128 bytes of a packet as categorical features.

`PathMiner` explores the execution tree using SE. Its SE engine executes the binary with a symbolic packet as input. Whenever it reaches a new node in the execution tree, it pushes the path constraints of the new node to a queue for the GA engine. The GA engine cycles through the path constraints and uses each one to set up a *path restricted* GA. In each GA, the chromosomes and objective function are customized to pressure convergence on packets that invoke long execution paths *and* satisfy the input path constraints.

`PathMiner` applies the GAs to two types of execution nodes. First, execution nodes in parts of the tree where SE has stopped making progress due to complex constraints. `PathMiner` mines these nodes for seed packets that the SE engine can execute concretely, i.e., without symbolic analysis. This adds new nodes to unexplored parts of the execution tree without invoking the expensive solver, since the concrete seeds prove the feasibility of their execution paths.

`PathMiner` uses GA to mine packets that match *leaf nodes*, which represent complete execution paths through the packet processor, for model building. `PathMiner` labels these packets with their execution path IDs and average execution time, as determined by the SE engine and GA. It trains

Path Constraint	
(Packet is IP) AND *(has valid IP Checksum)*	
Chromosome	
`pkt[0]:pkt[11]`	`= < any values >`
`pkt[12]`	`= 0x08`
`pkt[13]`	`= 0x00`
`pkt[14]:pkt[end]`	`= < any values >`

TABLE I. EXAMPLE OF A PATH CONSTRAINT AND THE GA CHROMOSOME THAT PATHMINER GENERATES.

supervised ML models with the labeled packets and returns the models.

A. Symbolic Execution Engine

`PathMiner` uses `S2E` [5] for symbolic execution. We chose `S2E` because it had two important features. First, it supports concrete execution, which lets it take advantage of the sample packets generated by the GAs. Second, `S2E` supports symbolic execution of binaries, which run in a `QEMU` VM with symbolic extensions. We extended `S2E` with a custom plugin to communicate with other components of `PathMiner` that exports path constraints as they are discovered and imports seed packets. We also configure `S2E` to timeout the constraint solver after 10 seconds. When a timeout happens, `S2E` selects a next branch by setting the packet to a random concrete value and taking whichever branch it invokes.

B. Path-Restricted Genetic Algorithms

`PathMiner` uses GAs to generate packets that invoke long execution paths and satisfy the constraint formulas of specific nodes in the execution tree. The key is deriving a *chromosome*, *fitness function*, and *repair function* from a specific constraint formula to pressure a GA towards generating packets that satisfy it.

Constraint-based Chromosome. `PathMiner` sets the chromosome of each GA to restrict it to a search space that is an over-approximation of the input constraint formula. The chromosome prohibits the GA from setting any byte to a value that make the packet unsatisfiable *independent of any other byte values*. This greatly reduces the search space of the GA because many execution paths in packet processors are gated by specific values appearing in specific packet header fields. Table I shows an example of a path constraint and the corresponding chromosome that `PathMiner` produces. The example constraint formula requires that a packet is a valid `IP` packet, which requires that bytes 12 and 13 (the Ethernet type field) are set to `0x08` and `0x00`, and also that the `IP` checksum is valid. The automatically generated chromosome eliminates packets that do not have the correct Ethernet type, but does not encode the more complex checksum constraint.

To compute the chromosome, `PathMiner` uses the Z3 [6] SMT solver. For each possible value of each byte, it checks whether the path constraint is still valid if a condition is appended to it requiring the byte to be set to that value. The generation is not expensive because each check is a small incremental change to the input path constraint formula, whose solution is cached in the solver.

Constraint-based Fitness. `PathMiner` uses a fitness function that captures the extent to which packets in the population satisfy the constraint formula. Equation 1 shows the fitness score in terms of the number of path constraints the packet satisfies (S) and the average execution time for the packet (E). The terms are normalized by the total number of constraints (C) and the maximum execution time observed across all the GAs so far ($\hat{max}(E)$).

$$F = \frac{S}{C} + \frac{E}{\hat{max}(E)} \tag{1}$$

The first term pressures evolution towards packets that satisfy the constraints. The second term biases it towards packets that cause high execution times.

`PathMiner` checks if a packet satisfies the constraint formula using a SMT solver (Z3 [6]) and, if it does not, requests that the solver return a *minimum unsatisfiable core*, or the smallest set of constraints that cannot be satisfied.

Constraint-based Repair. Finally, `PathMiner` also explicitly repairs some of the packets that do not satisfy the constraints. This has two purposes. First, it prevents unsatisfying packets with long execution times from dominating the search. Second, it seeds the GA with examples of packets that satisfy complex constraints to accelerate convergence. `PathMiner` only needs to repair a small fraction of the unsatisfying packets (*i.e.* $< 10\%$) because satisfying packets have high fitness scores, which causes them to be highly represented in subsequent generations.

Other Genetic Operators. `PathMiner` uses standard parameters for other genetic operators: a *two point crossover*, *tournament selector*, and *random replacement mutator*. These operators are widely used and more details can be found in the literature [18], [13], [30].

C. Supervised Learning

`PathMiner` trains ML models with the packets that GAs generate from complete execution paths. The models predict the execution time and path for each packet, using the first 128 bytes of the packet as features. Currently, `PathMiner` trains two types of models. First, a regression model that predicts how long the packet processor will take to execute a packet, in cycles. Second, a classification model that predicts which execution path the packet processor will invoke for the packet.

The current prototype uses a random forest of decision trees, with the number of trees fixed to 40. We chose the random forest because of minimal tuning parameters and the ability to handle packets as categorical data.

IV. PROFILING AN IP ROUTER

For a preliminary evaluation of `PathMiner`, we used it to profile a canonical `IP` router implemented with Click [22]. Figure 4 shows a block diagram of the router. Each block represents a Click module that is between 100 - 1200 lines of C++ code and implements a different router function, e.g., `IP` forwarding, `ICMP` processing, checksum validation,

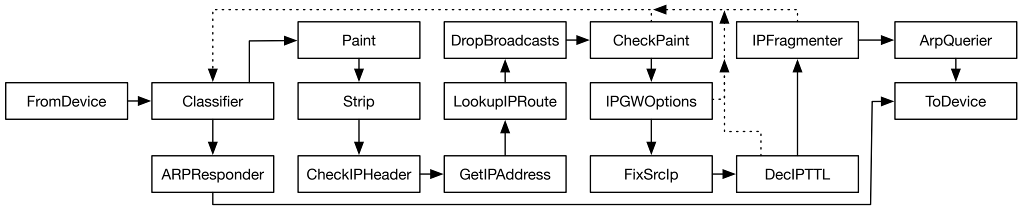

Fig. 4. Module-level control flow in the Click IP router.

Analysis System	S2E	PathMiner
Coverage Metric		
Execution Paths (*# Paths*)	368	2122
Execution Tree Size (*# Nodes*)	18677	70630
Longest Path Found (*# Cycles*)	10109	14449
Branches Covered (*# Branches Instructions*)	335	4480

TABLE II. COVERAGE STATISTICS FOR CLICK IP ROUTER.

or ARP processing. There are hundreds of Click modules and applications [23]. We chose the IP router because it is a practical and widely used application that has complex execution paths, such as for IP option processing, which have proven difficult for symbolic execution to handle without code annotations or modeling [7].

We ran a 12 hour analysis of the router with PathMiner, set to generate 1000 model training packets for each discovered execution path. As a baseline, we compared against a 12 hour analysis with S2E, using the packets generated by its constraint solver to train the same type of models that PathMiner produces. For a testing set, we collected all of the paths found by either profiler and used constrained GAs to produce 1000 new packets for each path.

The high level goal of was to understand how delay, i.e., execution time, varies across the router's execution paths and evaluate the models that PathMiner generated for predicting it. We sought to answer two main questions:

- Does PathMiner achieve higher coverage than SE? Can it find a tighter bound for worst-case delay?
- How accurately can PathMiner models predict delays and execution paths?

Coverage. Table IV summarizes coverage statistics for the PathMiner and S2E analyses. PathMiner discovered 5.76X more execution paths and covered 13.37X more branch instructions in the control flow of the router.

PathMiner also found longer paths, which are important to identify for DoS defense and scheduling in soft real-time systems. PathMiner's longest discovered path was for processing a fragmented packet with ttl = 0 and multiple options. As Table IV shows, it was nearly 50% higher than the longest path that S2E discovered, which lacked the ttl = 0 that caused expensive ICMP processing.

Predicting Delay. Figure 5 shows the distribution of error for predicting execution times when the training and testing sets included packets from mutually exclusive sets of paths. The histogram summarizes 10 trials. In each trial, 100 execution paths were withheld from the training data and only used for testing.

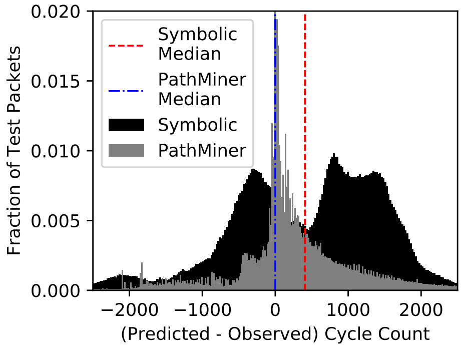

Fig. 5. Prediction error of PathMiner latency models for the IP router.

PathMiner's median error was 0 cycles. For 44.13% of packets, it predicted execution time to within 10 cycles. The distribution was unimodal and centered at 0, but skewed towards overestimating cycle count. In comparison, S2E's error distribution was bimodal with a median error of 407.33 cycles. It only predicted the execution times of 0.59% of packets to within 10 cycles. We discovered three factors that made PathMiner more effective than S2E alone.

First, it had higher code and execution path coverage. As Figure 6 shows, PathMiner found new execution paths at a linear rate throughout the analysis because feedback from the GAs continuously guided the SE to deeper execution paths without requiring it to solve complex constraints. S2E by itself, however, stalled after around 1.5 hours due to the issues described in Section II.

Second, PathMiner also generated more training samples throughout the analysis. Figure 7 shows that training samples are important for accurate models, especially in combination with high execution path coverage. Each point in Figure 7 represents 10 trials in which we measured absolute delay prediction error for packets from 100 random execution paths, using models trained with the first 400 - 1200 non-testing paths that PathMiner discovered.

Third, PathMiner also generated more diverse training samples for each path. Figure 8 plots the edit distance between successively generated training packets for one execution path. PathMiner's training samples had much higher edit distances, indicating that they better represented the range of possible packets that could invoke the execution path.

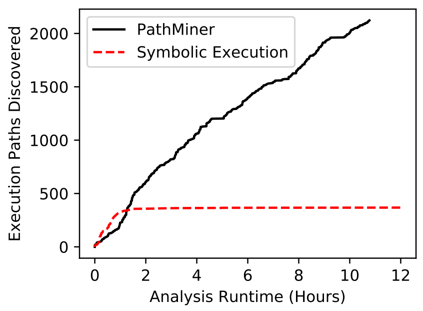

Fig. 6. Execution path discovery rate for `PathMiner` and `S2E`.

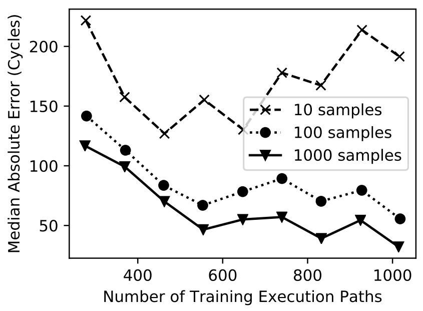

Fig. 7. Delay prediction error as number of training paths and samples per path varies.

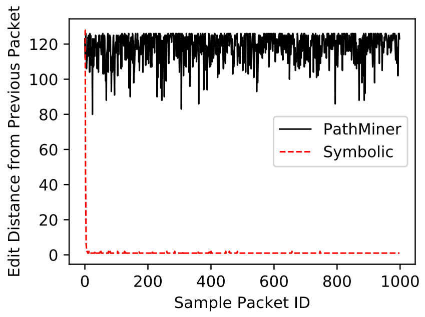

Fig. 8. Diversity of generated training samples for one execution path.

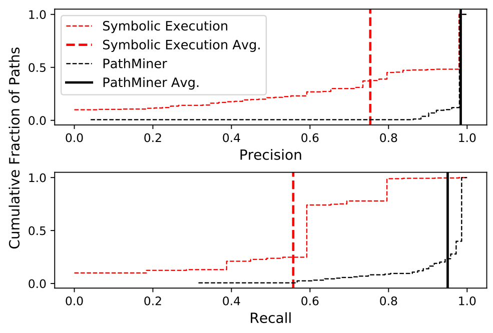

Fig. 9. Precision and Recall when predicting execution path.

Predicting Execution Path. Figure 9 shows that `PathMiner`'s models are also effective at predicting the *exact execution path* a packet will invoke, given it is a path that the models have been trained on. The figure shows precision and recall rates for predicting execution path based on packet header values, when the models were trained with packets from all discovered execution paths. `PathMiner` had an average precision of 0.96 and recall of 0.92. It identified the packets that belonged to most execution paths with *perfect* precision and recall. In comparison, `S2E` models had an average precision of 0.74 and a recall of 0.58.

V. Discussion and Future Work

Initial results with the `PathMiner` prototype are promising. There are many interesting questions to address in future work. In this section, we discuss three important topics of our active research.

Classification Throughput. `PathMiner`'s models are based on decision trees. We have not measured the throughput or latency of these models with our current prototype. However, performance should be high when using the appropriate platforms. Previous work has demonstrated that decision trees can be optimized for network traffic classification and sustain throughputs of over 90 million packets per second on a single commodity servers [33].

Networking Environments. `PathMiner` does not currently profile the Linux kernel's networking stack – it injects the symbolic packets directly into the packet processing binary. With the appropriate hooks in `S2E`, it is possible to inject symbolic packets at a lower level, e.g., into a virtual ethernet interface connected to the binary.

Related Work. Outside of the networking domain, there are other systems that augment symbolic execution with other complementary search heuristics [34], [12], [9], [16], [31]. These systems are designed for vulnerability finding; we have not evaluated whether they are also useful for delay predictability in software packet processors. `PathMiner` builds on all of these systems by introducing the concept of using GAs to efficiently generate randomized samples that match an SMT constraint by carefully configuring its genetic operations.

`PathMiner` also demonstrates that GAs can also be used to work around complex path constraints and increase the *depth* to the execution tree. This is an orthogonal approach to other recent systems that use GAs and SE, such as Driller [31]. The GAs in Driller fuzz small components of the program to increase the breadth of the execution tree, rather than depth.

VI. Conclusion

`PathMiner` takes an important step towards reducing the delay predictability gap between hardware and software packet processors. It analyzes binaries to automatically generate models of the execution times and paths that packets will invoke in software packet processors. `PathMiner` combines two powerful and complementary search techniques: *symbolic execution* and *genetic algorithms*. Symbolic execution provides a framework that allows `PathMiner` to methodically search a packet processor's execution paths, while customized genetic algorithms accelerate the symbolic execution and generate training data for building models. We evaluated `PathMiner` by using it to profile a software `IP` router. It produced highly predictive models and covered more execution paths than symbolic execution, without requiring code annotation or other user input. There are many use cases for `PathMiner` as a networking tool. The preliminary results motivate further exploration of its possible applications in both networking and as a general tool for binary analysis.

Acknowledgements: We thank the anonymous reviewers for their input on this paper. This research was partially supported by NSF grant number 1406225 and ONR grant number N00014-15-1-2006.

References

[1] "Data plane development kit," *URL http://dpdk. org*.

[2] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[3] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A NICE way to test OpenFlow applications," in *Proc. Network System Design and Implementation (NSDI)*, Apr. 2012.

[4] A. Chen, A. Sriraman, T. Vaidya, Y. Zhang, A. Haeberlen, B. T. Loo, L. T. X. Phan, M. Sherr, C. Shields, and W. Zhou, "Dispersing asymmetric ddos attacks with splitstack." in *HotNets*, 2016, pp. 197–203.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[6] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, 2008.

[7] M. Dobrescu and K. Argyraki, "Software dataplane verification," *Communications of the ACM*, vol. 58, no. 11, pp. 113–121, 2015.

[8] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks: exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 15–28.

[9] W. Drewry and T. Ormandy, "Flayer: Exposing application internals." *WOOT*, vol. 7, pp. 1–9, 2007.

[10] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.

[11] S. Forrest and M. Mitchell, "What makes a problem hard for a genetic algorithm? some anomalous results and their explanation," *Machine Learning*, vol. 13, no. 2-3, pp. 285–319, 1993.

[12] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 474–484.

[13] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning, 1989," *Reading: Addison-Wesley*, 1989.

[14] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Satisfiability solvers," *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[15] V. Gurevich, "Programmable data plane at terabit speeds," http://open-nfp.org/static/app/pdfs/Sponsor-Lecture-Vladimir-Data-Plane-Acceleration-at-Terabit-Speeds.pdf.

[16] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations." in *USENIX Security Symposium*, 2013, pp. 49–64.

[17] J. H. Holland, "Adaptation in natural and artificial systems. an introductory analysis with application to biology, control, and artificial intelligence," *Ann Arbor, MI: University of Michigan Press*, 1975.

[18] ——, *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[19] Q. Hou, C. Qiu, K. Mu, Q. Qi, and Y. Lu, "A cloud gaming system based on nvidia grid gpu," in *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2014 13th International Symposium on*. IEEE, 2014, pp. 73–77.

[20] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 9.

[21] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[22] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug 2000.

[23] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[25] C. S. Perone, "Pyevolve: a python open-source framework for genetic algorithms," *Acm Sigevolution*, vol. 4, no. 1, pp. 12–20, 2009.

[26] L. T. X. Phan, "Real-time network function virtualization using timing interfaces," in *International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, 2016.

[27] P. Puschner and R. Nossal, "Testing the results of static worst-case execution-time analysis," in *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE, 1998, pp. 134–143.

[28] L. Rizzo and M. Landi, "Netmap: Memory Mapped Access to Network Devices," in *Proc. ACM SIGCOMM*, 2011.

[29] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese+, H. Balakrishnan, M. Alizadeh, and N. McKeown, "Packet transactions: High-level programming for line-rate switches."

[30] M. Srinivas and L. M. Patnaik, "Genetic algorithms: A survey," *computer*, vol. 27, no. 6, pp. 17–26, 1994.

[31] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[32] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "Symnet: scalable symbolic execution for modern networks," in *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*. ACM, 2016, pp. 314–327.

[33] D. Tong, Y. R. Qu, and V. K. Prasanna, "High-throughput traffic classification on multi-core processors," in *2014 IEEE 15th International Conference on High Performance Switching and Routing (HPSR)*, July 2014, pp. 138–145.

[34] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 497–512.

[35] J. Wegener and F. Mueller, "A comparison of static analysis and evolutionary testing for the verification of timing constraints," *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001.

[36] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, and O. Sokolsky, "Rt-open stack: Cpu resource management for real-time cloud computing," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE, 2015, pp. 179–186.

[37] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified nat," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 141–154.