Detecting Routing Loops in the Data Plane

Jan Kučera
CESNET
jan.kucera@cesnet.cz

Ran Ben Basat
Harvard University and UCL
ran@seas.harvard.edu

Mário Kuka
CESNET
mario.kuka@cesnet.cz

Gianni Antichi
Queen Mary University of London
g.antichi@qmul.ac.uk

Minlan Yu
Harvard University
minlanyu@seas.harvard.edu

Michael Mitzenmacher
Harvard University
michaelm@eecs.harvard.edu

ABSTRACT

Routing loops can harm network operation. Existing loop detection mechanisms, including mirroring packets, storing state on switches, or encoding the path onto packets, impose significant overheads on either the switches or the network.

We present Unroller, a solution that enables real-time identification of routing loops in the data plane with minimal overheads. Our algorithms encode a varying fixed-size subset of the traversed path on each packet. That way, our overhead is independent of the path length, while we can detect the loop once the packet returns to some encoded switch. We implemented Unroller in P4 and compiled into three different FPGA targets. We then compared it against state-of-the-art solutions on real WAN and data center topologies and show that it requires from 6x to 100x fewer bits added to packets than existing methods.

CCS CONCEPTS

• **Networks → Network algorithms**; **Network monitoring**; *Programmable networks*; *In-network processing*;

KEYWORDS

Network algorithms, routing loops, programmable data planes.

ACM Reference Format:
Jan Kučera, Ran Ben Basat, Mário Kuka, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. 2020. Detecting Routing Loops in the Data Plane. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3386367.3431303

1 INTRODUCTION

Real-time detection of traffic loops is essential for the performance of today's networks. Unidentified loops may lead to losses, which in turn increase the tail latency [14]. Also, packet losses due to traffic loops are often interpreted as a signal of congestion, e.g., in TCP, leading to a reduction in throughput [1]. As an example, in a production cluster of 2500 switches, Microsoft reported that traffic trapped in routing loops led to a significant increase in overall traffic [29]. Also, it has been demonstrated that the portion of traffic not caught in the loop but sharing some of the affected links can be severely affected in terms of delay and jitter [14]. Finally, loops are one of the main causes for routing instability which can affect performance on the network as a whole [10, 25, 26]

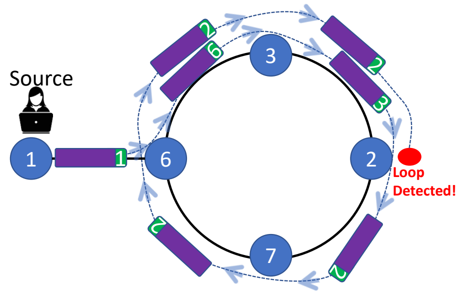


Figure 1: Unroller in action: each packet stores the minimal switch ID seen and resets the stored ID after each phase. When a packet reaches the switch with the stored ID, the loop is reported.

Advanced approaches to loop detection include storing state on switches [19, 24] or mirroring just selected packet header fields [9, 13]. The former leads to significant overhead on switch memory, while the latter leads to significant overhead on the network. Both aspects are important, as the scarce switch SRAM memory can be instead used for ACL rules or customized forwarding [23], while excessive control traffic can have prohibitive data collection overheads [21]. Lately, the recent advances of programmable switches [5] has opened the opportunity to tackle the routing loop detection problem by storing path information directly on packets. For example, the in-network telemetry (INT) [11] allows each switch to put its ID on a packet as it passes by. As a consequence, if a switch sees its own ID on an incoming packet, it can conclude the existence of a routing loop and take appropriate action, e.g., report and reroute the packet. This simple solution suffers from an important drawback: storing the full path information on a packet takes significant header space. For a path of six hops, for example, we need 32 Bytes (8 Byte INT header and 4 Byte switch ID for each hop) [11], which is an overhead of 3.2% for packets with an average size of 1 KBytes. While a reduced overhead can be obtained for specific data center topologies [27], a more generic approach is needed when dealing with any arbitrary big topology.

In this paper, we design Unroller, a solution that enables real-time identification of traffic loops while keeping low overhead on both network and switches. The idea is to store within each packet only *a subset* of the path taken, even a single switch ID, while *guaranteeing* that a routing loop can be identified in a bounded number of hops. This is possible by dividing the path of a packet in phases, i.e., consecutive series of hops, that increases exponentially, i.e., $1, 2, 4, 8, \ldots$. Every time a switch processes a new packet, it is allowed to store its ID only if it is smaller than the one currently stored into the packet or if we are at the beginning of a new phase. Intuitively, not too long after reaching the loop, the packet will enter a phase that is long enough to reach again the last switch that has updated its ID on the packet (see Figure 1). We evaluate Unroller against state-of-the-art solutions using real WAN and data center topologies. We show that, although storing only partial information into packets might introduce errors, the probability of falsely reporting a loop is negligible in practice. Furthermore, our solution requires from 6x to 100x less bits added to packets than existing methods.

In summary, the main contributions of this paper are:

- We present Unroller, a novel solution that enables real-time identification of routing loops in the data plane by storing information on packets.
- We evaluate Unroller on a number of real WAN and data center topologies. We also implement our solution in P4 and compile into three different FPGA targets.
- We analyze Unroller and rigorously prove performance bounds.
- We open source our code: https://github.com/kucejan/unroller.

2 DESIGN SPACE

Past proposals can be classified into three main categories depending on how they handle the information needed for detecting loops; (1) keep flow state at switches; (2) mirror information at switches; or (3) keep information on packets.

Storing flow information at switches [18, 24] and periodically exporting the information to a collector can put too much pressure on the switch hardware. Indeed, keeping state for a large number of active flows (e.g., up to 100K [22]) is by itself challenging with limited switch space (e.g., 100 MB [20]). Moreover, space is at a premium because operators need the memory for more essential control functions such as ACL rules, customized forwarding [23], and other network functions and applications [16, 20]. The advantage of storing information on switches is the low network overhead; we only need to occasionally export the switches' state for analysis and can avoid using excessive control bandwidth.

Mirroring information at switches upon a packet arrival [13] or after a given timeout [9] creates significant scalability concerns for both trace collection and analysis. The traffic in a large-scale data center network equipped with hundreds of thousands of servers can introduce terabits of traffic [12, 22]. Assuming a CPU core can process tracing traffic at 10 Gbps, on the order of thousands of CPU cores would be required for trace analysis [29], which is prohibitively expensive.

Finally, a third set of solutions propose to keep information on packets [11, 15, 27]. Specifically both INT [11] and Tiny Program Packets [15] suggest a mechanism for each switch to record their ID in the incoming packet. This allows rapid detection of loops in the data plane[1] at the cost of a per packet overhead that grows linearly with the network diameter, i.e, the ID is encoded in 4B and 2B for INT and TPP respectively. With Pathdump [27], the authors instead enable on-line routing loop detection by leveraging the fact that commodity SDN switches can recognize only two VLAN tags in hardware. With this in mind, they consider only scenarios where a third VLAN tag only arises in the presence of a loop, and when an attempt is made to add a third tag, the switch CPU is invoked to manage the loop detection.

[1]If a switch receives a packet with its ID already stored, then most likely the packet has entered in a loop.

Table 1: Comparisons of Unroller and the state-of-the-art solutions for routing loop detection.

Type	Solution	Real Time	Switch Overhead	Network Overhead
On-switch State	FlowRadar [18] Hash IP Traceb. [24]	✗	high	low
Header Mirroring	NetSight [13] Everflow [29] Trajectory Samp. [9]	✗	low	high
Full Path Encoding on Packets	INT [11] TPP [15] PathDump [27]	✓	low	high
Partial Encoding	**Unroller**	✓	**low**	**low**

A last key classification for such algorithms is whether they can detect a loop *in real time*: while a packet is in flight. Real-time detection of loops enables (1) selective reporting: let the packet traverse the loop again to record the identifiers of the participating switches; and (2) active rerouting: forward the packet to a different port in an attempt to avoid packet loss. All existing solutions are either unable to detect loops in real time or have a packet overhead that is linear in the number of hops, as presented in Table 1. Given the design space with the trade-offs current solutions face, in this paper, we answer the following question:

Can we design an algorithm that detects routing loops at real time, in the data plane, while keeping low switch and network overheads?

We show that this is possible with Unroller, a technique to encode only a small subset, e.g., a single identifier, of the switches ID along the path, while *guaranteeing* detection in a bounded number of hops.

3 UNROLLER

One possible approach to detect loops by encoding information onto packets is to store the identifier of all switches that the packet traverses. This is how INT would handle this task. When a switch receives a packet, it checks if it is on the packet's list and, if so, reports a loop. As previously discussed, this generally adds significant bandwidth overhead and should be avoided.

A possible alternative is to store a Bloom filter which encodes the set of visited switches. Intuitively, we can hold a compressed representation of the path and save bandwidth at the cost of false positives. As before, once a switch is reported as positive by the filter, we report a loop. This Bloom filter solution must deal with false positives, but it remains wasteful even without that issue. Intuitively, there is no need to remember *all* switches on the loop, but only *some switch* on the loop. If a packet stores the same switch ID while traversing the entire loop, we can report the loop when we see the repeated switch ID. Let us first assume that the packet's first hop is already part of the loop. In this case, we can record on the packet

Table 2: List of symbols and notations.

Symbol	Definition
B	The number of hops before the loop.
L	The number of switches in the loop.
X	The number of hops before reaching a switch twice (B+L).
b	The phase growth base; the i'th phase lasts for b^i hops.
z	The Unroller bit-overhead on each packet.
c	The number of switch IDs encoded into packets.
H	The number of hash functions used on each switch ID.
Th	The threshold for reporting a loop.

the *minimum* switch ID that it has seen. We are guaranteed to detect the loop after two iterations through the loop; the minimal switch is recorded in the first loop and observed again in the second loop. The problem becomes a bit more complicated when there could be a path of switches the packet traverses before reaching the loop. In this case, the above approach would fail when the minimal identifier appears on the path leading to the loop rather than the loop itself. We suggest the following solution (Table 2 summarizes the notation used in this paper).

Let B be the number of hops *before* the loop and L be the number of switches in the loop. Notice that any algorithm would require the packet to traverse $X \triangleq B + L$ hops before reaching some switch for the second time, which gives a lower bound on the number of hops required for detection. We now show a deterministic algorithm that stores a single switch ID, has no false positives, and finds the loop after at most $4.67X$ hops (without knowing B or L). As before, we keep the minimum identifier we have seen, but now we occasionally *reset* the identifier as though we are restarting, and we gradually increase the resetting intervals.

Our algorithm has a parameter b that determines how aggressively we increase the resetting intervals. The execution takes place in *phases* so that at the end of each phase, we reset the stored identifier; the i'th phase lasts for b^i hops. We prove that after no more than

$$(2L-1)+\max\left\{\frac{2bL-1}{b-1},\, bB+1\right\} \le 4.67X$$

hops (the inequality holds for $b=4$), the packet reaches a switch that can report the loop. If the switches can perform floating point operations, or if we can compute $\lfloor b^i \rfloor$ for non-integer b using a lookup table, it is possible to optimize the ratio further.

THEOREM 1. *Our algorithm identifies the loop after at most* $(2L-1)+\max\left\{\frac{2bL-1}{b-1}, bB+1\right\}$ *hops* at the worst case.

We split the proof of the theorem into three simple lemmas.

LEMMA 2. *After at most* $\frac{2bL-1}{b-1}$ *hops, we get to a phase that lasts at least* $2L$ *hops.*

PROOF. Let us first denote by p the first phase number that lasts for at least $2L$ hops. Observe that $p = \lceil \log_b 2L \rceil$; the number of hops until we reach this phase is then

$$\sum_{i=0}^{p-1} b^i = \frac{b^p-1}{b-1} \le \frac{2bL-1}{b-1}. \qquad \square$$

LEMMA 3. *After at most* $bB+1$ *hops, the stored ID is from a switch on the loop.*

PROOF. We know that after B hops the packet reaches the loop. We want to show that, once the packet reaches the first switch in the loop, after at most $(b-1)B+1$ additional hops the phase ends and the identifier resets, at which point the stored ID will be from a switch on the loop. Since the previous phase (if one exists) before reaching the first switch in the loop cannot last more than B hops, it follows that the current one must end within $b \cdot B$ hops. A slightly tighter analysis shows that it actually ends within $(b-1)B+1$ additional hops. Denote the phase number when we reach the first switch on the loop by p. By the end of this phase, the stored ID will be from a switch in the loop. We have that

$$\frac{b^p-1}{b-1} = \sum_{i=0}^{p-1} b^i < B,$$

and thus $p < \log_b(B(b-1)+1)$. As the current phase is of length b^p, the lemma follows. $\square$

LEMMA 4. *If at the start of a phase the stored identifier is from a switch on the loop and the phase lasts at least* $2L-1$ *hops, then we terminate after at most* $2L-1$ *hops.*

PROOF. Let v be the switch with the smallest ID in the loop. From (1) and (2) it follows that (i) the packet has already reached the loop, (ii) that the ID that is stored of a node in the loop and that (iii) the phase is long enough; after at most $L-1$ hops the packet reaches v and thereafter does not change the stored identifier. After another L hops it reaches v again and the loop is reported. $\square$

3.1 Lower Bound

Our algorithm only guarantees detection after $4.67X$ hops. An interesting question is *what is the minimal number of hops required for loop detection by an algorithm that stores a single identifier*? As we now state, deferring the details to Appendix A, any deterministic algorithm that does not assume knowledge of B requires at least $\approx 3.73X$ hops detection time. This shows that our approach is not far from optimal for deterministic algorithms.

THEOREM 5. *Any deterministic loop detection algorithm that stores a single identifier requires at least* $3.73X \cdot (1-o(1))$ *hops for detection in the worst case.*

3.2 Average Case Analysis

The above analysis shows that we require at most 4.67 times as many hops to report a loop than the costly algorithm that stores the entire path. This analysis holds at the worst case, but it is also useful to analyze the *average case*. For reasoning about the average case, we require that the switch IDs will be random so that each switch has the same probability of holding the smallest ID. If this is not the case, we can use hashed switch IDs for the algorithm; these may introduce false positives (similar to the Bloom filter algorithm), but the trade-off between overhead to error is much more favorable in our algorithm. Alternatively, we may consider a random *permutation* on the switch identifiers that is known to all switches. We have no false negatives and all loops are still guaranteed to be reported. We show here that in the average case, the loop is detected after at most $3X$ hops, when $b=3$. There are three cases to consider, depending on the length, denoted q, of the first phase that begins on the loop with length at least L.

If $q=(1+\alpha)L$ for some $0 \le \alpha \le 1$, then up to this phase, by our previous analysis, the packet has traversed at most $(q-1)/(b-1)$ hops. In this phase, since the switch with the minimal identifier is equally likely to be any on the loop, we hit the switch with the minimal identifier twice with probability α, and in this case, the expected number of additional hops is $(1+\alpha/2)L$. If the switch with the minimal identifier is not hit twice in this phase, which occurs with probability $(1-\alpha)$, it will be hit twice in the next phase, after an expected $(1+\alpha)L+(1+(1-\alpha)/2)L$ hops. Overall, the total expected number of hops is at most

$$L\left(\frac{1+\alpha}{b-1}+2-\frac{\alpha^2}{2}+\frac{(1-\alpha)^2}{2}\right)=L\left(\frac{1+\alpha}{b-1}+2.5-\alpha\right).$$

For $b=3$, this expression is at most $3L$.

If $2L<q\leq bL$, then the loop will be found in this phase, after an expected $3L/2$ hops. Up to this point, the packet has traversed at most $bL/(b-1)$ hops, which is also $3L/2$ when $b=3$, giving an overall count of at most $3L$ hops.

If $q>bL$, then the previous phase was at least L hops but did not start on the loop. In this case, we have traversed at most $bB+1$ hops, and B is at least $L/(b-1)$. The loop will be found in this phase, after an expected $3L/2$ hops. In this case, X is at least $B+L$, and the expected number of hops to find the loop is at most $bB+1+1.5L\leq 3X$.

For $b=3$, in all cases, we have shown that after at most $3X$ hops we identify the loop, and this is the best choice for b for the average case analysis. The average case analysis provides a different bound than the lower bound for the worst-case analysis, and uses a different choice of b than our best upper bound for the worst-case analysis.

3.3 Reducing the Per-Packet Overhead

The above algorithms suggest storing a switch identifier on each packet. However, in some cases, the identifiers may be large and pose an undesirable overhead. In such cases we propose to *hash* the switch identifiers into z bits. That is, instead of storing a switch identifier, Unroller will encode its smaller hash onto the packet. This reduces the number of bits added to each packet but introduces false positives as two switches not on a loop, but simply on the path, may have the same hash.

We propose a simple counting technique that exponentially reduces the probability of false positives. We add a small counter that tracks the number of times we have seen a switch whose hash matches the one on the packet. Once the counter reaches a predetermined threshold of Th, we report the loop. If there is a loop, the counter eventually reaches Th; if there is no loop, a false positive now requires Th switches on the path to have the same hash, which is much more unlikely. This solution requires an additional $\lceil\log_2 Th\rceil$ bits per packet[2], but significantly reduces the chance of false reporting. For example, on a path of length 20 hops, with $Th=4$, $z=7$, and $b=4$, the chance of false positives is lower than 10^{-5} while using $(7+2)$ bits of overhead per packet. Therefore, we can run with only a few false positives while reducing the overhead by 72%. We note that using $Th>1$ does not come for free as it increases the number of hops required for detection (namely, by $(Th-1)\cdot L$ hops).

3.4 Trading Bandwidth for Convergence

So far, we have allowed the algorithm to store a single identifier. As we saw, this allows us to derive algorithms that are 3-4.67 times slower than the X hops lower bound (which assumes no bandwidth constraints). A natural question is whether we can get faster detection if we allow storing more than one identifier but not the entire path.

[2] We do not need to encode the value Th but report the hop that sees a hash match when the counter equals Th-1.

Table 3: Parameters encoded in the packets' header being used by our algorithm.

Values encoded in packet headers	
X_{cnt}[3]	The current number of visited hops of the packet along its path.
$SW_{ids}[]$	The array of the current switch IDs seen.
Th_{cnt}	The current value of the threshold counter.

The main drawback of storing just one identifier is that we had to balance the rate in which we increase the reset time (the parameter b) in a way that we do not lose much when $B\gg L$ but also when $L\gg B$.

In Appendix B, we explore how to use multiple identifiers on packets to reduce the expected number of hops required for detection. Specifically, we show that by using H hash functions and storing c identifiers for each (a total of $c\cdot H$ identifiers), we can reduce significantly the number of hops. Intuitively, using multiple hashes allows different switches to have "minimum IDs" with respect to some hash function while each of the c identifier stored for a hash function tracks the minimum only on a $1/c$-fraction of the phase.

3.5 Discussion

Here, we discuss the importance of phases and the trade-off associated with the identification of the switches involved in the loop.

Importance of switch ID resetting. Let us assume a variant of Unroller where each switch inserts its ID, with a set probability, only if the incoming packet does not already carry the maximum number of IDs. This solution works well when the packet's first hop is already part of the loop. If, however, the packet encounters a number of hops before the loop, this solution might introduce false negatives when only the pre-loop IDs are stored within the packet. By introducing phases in the algorithm, we force the values already stored within the packet to be overwritten at times, thus avoiding this problem.

Identification of switches involved in a loop. There is an obvious trade-off between the detection of the loop and the additional identification of all the switches involved. Directly recording as many IDs as possible into packets aids the discovery of network elements involved in the loop. However, this comes at the cost of additional overhead on packets, which leads, in normal conditions, to undesired effects on network performance [3]. With Unroller, we opted for a lightweight mechanism to detect loops. Once a loop is identified it is possible, for example, to tag the packet to collect the involved switch IDs and send a report for analysis.

4 IMPLEMENTATION

We implemented Unroller using the P4$_{16}$ language [8] and compiled on a software target BMv2 [7], and three FPGA based targets using the P4-To-VHDL compiler [4].

P4 implementation. The core of Unroller is implemented in 60 lines of code. The implementation consists of a single control block applied at the ingress pipeline. The input program parameters are b, z, c, H and Th (Table 2). Additionally, Unroller requires extra

[3] In cases where the hop number can be inferred from the TTL (e.g., see [2, 3]), we can avoid storing X_{cnt} and reduce the bit overhead.

Table 4: Architecture HW resources utilization results.

Platform	LUTs	REGs	BRAM	Frequency
Virtex 7	26 234 (7.23 %)	29 944 (4.13 %)	396 kb (1.17 %)	224 MHz
Virtex US+	26 221 (7.23 %)	30 520 (4.21 %)	684 kb (2.02 %)	225 MHz
Stratix 10	21 917 (1.17 %)	45 907 (1.22 %)	301 kb (0.12 %)	189 MHz

information being carried on each packet, as summarized in Table 3. Observe that X_{cnt} requires at most 8 bits [3], $SW_{ids}[]$ takes $c \cdot H \cdot z$ bits, while Th_{cnt} only needs $\log_2 Th$ bits Here, we assume that each switch has an unique identifier, stored in a register alongside all the aforementioned configuration parameters. No match-action tables are needed. All the logic fits in a single `apply` section of the control block. Specifically, for each incoming packet, we (1) read the configuration parameters from the registers and increment X$_{cnt}$; (2) evaluate the hash functions to randomize the switch ID; (3) check if one of the IDs ($SW_{ids}[]$) stored within the packet have to be updated; (4) drop the packet and inform the controller when a loop is identified. Unroller updates the IDs list present in the packet header only if either there is space for a new value, the current switch has an ID smaller than the one currently stored in the list, or if the packet enters a new phase. This is the case when the X_{cnt} counter stored within the incoming packet is equal to a power of the phase growth base b. Fortunately, for $b = 2$ or $b = 4$, this operation can be performed using standard bitwise checks. Th_{cnt} counter tracks the number of times the packet has seen a switch whose ID matches one of the stored values in the list. Once the counter reaches a predetermined threshold of $Th - 1$, a loop is reported.

Compiling Unroller to programmable switches. We compiled Unroller on the P4 software switch target based on the behavioral model (BMv2). Here, the main constraint is the number and pattern of accesses to on-chip registers [20]. To reduce the number of operations, we used a 256-sized lookup table that records, for each possible X_{cnt}, whether it is the start of a new phase. Alternatively, it is possible to store pre-hashed identifiers into registers, to reduce the number of hash operations. Unroller requires two pipeline stages, uses minimal resources, and does not store any per-flow state in the switch. Furthermore, if the set phase growth base b is not a power of two, a lookup table is necessary for determining the packet's phase. This is because specific operations such as division or power evaluation are not natively supported by hardware.

Compiling Unroller to FPGAs. We used the P4-To-VHDL compiler to port the produced P4 code to different FPGA chips. This was not a one-step process as the original code needed a few adaptations to meet FPGA timing constraints. Specifically, the compiler allows calling actions that manipulate packets only from a match-action table and not directly from a control block. Because of this, we added a dummy match-action table with a single default action unconditionally manipulating the packet. We compiled the Unroller logic into three different FPGA-based targets supporting 100GbE ports: Xilinx Virtex 7 (model XCVH580T), Xilinx UltraScale+ (model XCVU7P) and Intel Stratix 10 (model 1SG280HU). Table 4 shows the chip occupancy and the maximum frequency for all the platforms. Here, we can see that Unroller logic is lightweight, requiring less than 8% of chip resources. Since the synthesized architectures are fully pipelined, i.e., capable of processing a new packet every clock cycle, the frequency can be directly correlated with the maximum achievable throughput: ~220 Mpps for Xilinx devices, and ~190 Mpps for the Intel platform. This is more than 100 Gbps for minimum-sized Ethernet packets. Given the targets are dimensioned for 100GbE processing, we can fairly state that Unroller logic does not introduce any throughput degradation.

Table 5: Unroller vs. state-of-the-art solutions on real topologies.

Topology	# of Nodes	Diameter	PathDump	Bloom filter	Unroller	
			Overhead *(bits)*	Overhead *(bits)*	Avg Time *(#hops/X)*	Overhead *(bits)*
Stanford	16	2	×	171	1.74	25
BellSouth	51	7	×	189	1.56	25
GEANT	40	8	×	608	2.13	27
ATT-NA	25	5	×	608	2.15	27
UsCarrier	158	35	×	2466	2.47	28
FatTree4	20	4	64	414	1.73	28

5 EVALUATION

We evaluated Unroller with a Python simulator that generates paths based on the required number of hops before entering a loop (B) and the number of hops comprising the loop itself (L). Unless otherwise stated, each data point reflects 3M runs. Switch identifiers are randomly generated 32-bit numbers, and the default Unroller configuration parameters are $b = 4$ phase base, $c = 1$, $H = 1$ (one hash function) and $Th = 1$ reporting threshold (see Table 2 for notation description).

Comparing Unroller to state-of-the-art solutions. Here we used several real topologies, with different sizes, spanning from WAN to data centers [17, 28]. We compared the loop detection capabilities of Unroller against state-of-the-art solutions that work in real time: (1) PathDump [27] and (2) an especially crafted approach that adds a Bloom Filter into packets to store switch IDs. The former adds a fixed overhead on each packet, i.e., 64 bits, and does not experience false positives, but can only be applied to a very limited set of topologies [27], e.g., FatTree and VL2. By employing a probabilistic data structure to store switch IDs, the latter can introduce false positives, as Unroller does. To compare both solutions fairly, we randomly picked two nodes in each considered topology and selected a shortest path between them. Out of all possible loops that intersect with that path, we picked one uniformly at random. We then measured, over 3M runs, the minimum overhead (in bits) needed in each packet so that no false positives were reported. Table 5 shows the results. Unroller can detect loops, without experiencing any false positives, using a very small packet overhead. Depending on the topology, our solution requires from 6x to 100x fewer bits than the Bloom Filter counterpart. This comes at the expense of detection speed: while the Bloom Filter can identify a loop as soon as a switch is hit twice by the same packet, Unroller might require one or two extra passes over the loop, as reported in the *Avg Time* column in Table 5. INT would require packets to store an increasing number of switch IDs at each hop, making this approach more expensive (in terms of per-packet bit overhead) than those previously discussed.

Sensitivity analysis. Here, we aim at assessing how the different parameters introduced in this paper (see Table 2) might affect Unroller performance. Unless otherwise stated, the adopted default values are the following: $B = 5$ hops before the loop, $L = 20$ loop hops, $z = 32$ bits per packet, $c = 1$, $H = 1$ (one hash function) and $Th = 1$ reporting threshold. We first evaluate the average detection time, measured as the ratio between the number of hops required for detection and the $X = B + L$ hops lower bound. This time is affected by the loop

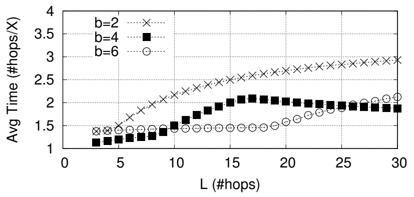


Figure 2: Detection time varying L and b.

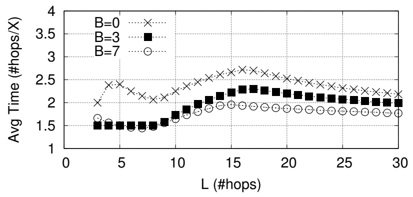


Figure 3: Detection time varying L and B.

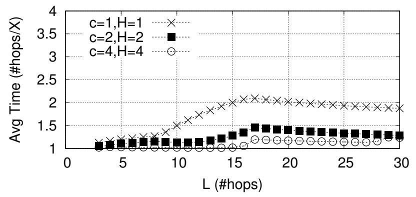
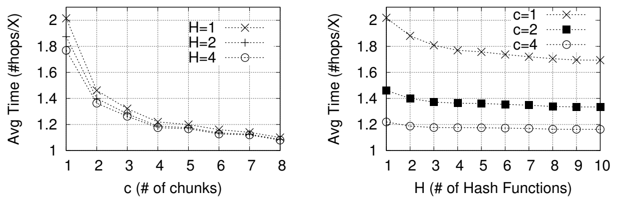


Figure 4: Detection time varying L and c, H.



(a) Varying c (b) Varying H

Figure 5: Detection time for different c, H configurations.

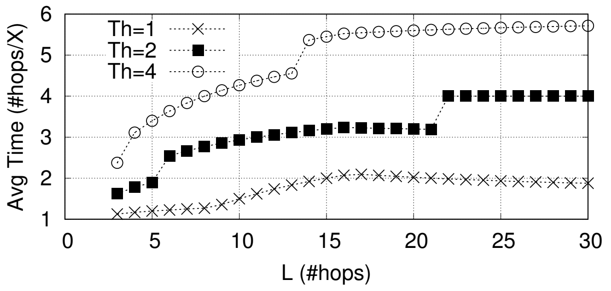
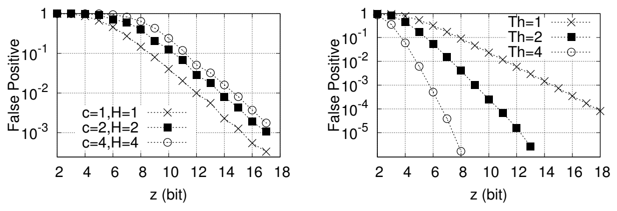


Figure 7: Detection time using counting technique varying Th.



(a) Varying c and H (b) Varying Th

Figure 6: False Positives when compressing switch IDs to z bits.

length L when storing only a single full switch ID in each packet. Figure 2 shows the relationship when varying the b parameter. The smaller the value of b, the more aggressively Unroller resets the switch ID stored in the packet, causing an increase of the average detection time. Figure 3 shows the impact of B when b is fixed to 4. Here, the average detection time increases when B decreases. This is the effect of the resetting interval mechanism.

In Figure 4, we fixed $b = 4$ and $B = 5$ and studied the effect on the average detection time of partitioning each phase into c chunks and randomizing the switch ID using H hash functions. Specifically, we stored $c \cdot H$ hashed switch identifiers into each packet. All of the stored IDs are compared to the current switch identifier, and a loop is reported if a match is found. Clearly, the more chunks and hashes used, the better it is for the average detection time. Figures 5(a) and 5(b) show in more details the individual impact of parameters c and H to the detection time. We can see that the improvement is greater when we are increasing the number of chunks c when compared to increasing the number of hashed switch identifiers H. This means that Unroller is more sensitive to the number of chunks rather than the number of hash functions.

Although storing multiple identifiers in the packet ($c > 1$ or $H > 1$) improves the average loop detection time, it also imposes a bigger overhead on each packet. Thus, we analyzed the extension of the algorithm, which reduces the per-packet overhead by compressing the switch identifiers into z-bit values. This practice, however, may introduce false positives. We test using a path length of 20 hops, with $B = 20$ and $L = 0$. As the adopted path does not contain loops, any reported loop by Unroller is a false positive. Figure 6(a) depicts the effect of the compression over the false positives when varying c and H and keeping the hash width z. Partitioning of phases ($c > 0$) or storing more hashed identifiers ($H > 0$), if combined with the compression, increases the false positive rate but leads to faster detection of loops (Figure 4).

Figure 6(b) shows similar results when the threshold technique to reduce the false positive rate is deployed. In this case, the false positives are reduced exponentially with the size of the threshold. However, this comes at the cost of a slight increase in the average detection time, as demonstrated in Figure 7.

6 CONCLUSION

In this paper, we presented Unroller, a lightweight loop detection solution that is readily deployable on emerging technologies such as programmable switches. We evaluated our solution and showed that it could quickly and accurately detect routing loops using a minimal bit-overhead on packets. Further, Unroller does not store state on switches, leaving their scarce memory to other applications. Unlike some of the existing solutions, Unroller can identify loops in real time, by the switches themselves and without a remote analysis node. We envision that such a capability would enable rerouting mechanisms that could prevent packet losses that happen when packets traverse a loop until their TTL zeros out. For example, recently introduced solutions to enable near-optimal compression of backup rules [6] can be adopted in cooperation with Unroller to quickly reroute packets on pre-determined backup ports upon the detection of a loop.

Acknowledgments. We thank our shepherd and anonymous reviewers for their valuable comments. This work was supported by the project Reg. No. CZ.02.1.01/0.0/0.0/16_013/0001797 co-funded by the Ministry of Education, Youth and Sports of the Czech Republic and European Regional Development Fund, by the Brno University of Technology under number FIT-S-20-6309, by NSF CCF-1563710, CCF-1563710, and CNS-1834263, the Zuckerman foundation and the UK's EPSRC under the projects EARL (EP/P025374/1) and NEAT (EP/T007206/1).

A LOWER BOUND

Observe that any such algorithm can be determined by the interval it takes before resetting the identifier ($b^0, b^1, \ldots$, in our algorithm). Let x_1 denote the number of hops before the first reset, x_2 denotes the number of hops before the second one, etc. We denote by $\beta \triangleq \max_{n\in\mathbb{N}} \left\{ \frac{x_n}{\sum_{i=1}^{n-1} x_i} \right\}$ the maximal growth rate in the resetting periods. Let n be such that $\frac{x_n}{\sum_{i=1}^{n-1} x_i} = \beta$ and denote $y = \sum_{i=1}^{n-1} x_i$.[4] We also note that by setting a minimal value for n (i.e., $\beta \triangleq \max_{n\in\mathbb{N}, n\geq T} \left\{ \frac{x_n}{\sum_{i=1}^{n-1} x_i} \right\}$ for some T) we get an arbitrarily large number of hops before possible detection (X). We start with some lemmas.

LEMMA 6. *Any deterministic algorithm that stores a single identifier must make at least $(\beta + 1) \cdot X - O(1)$ hops before identifying a loop.*

PROOF. Let $B = y + 1, L = 2$, and consider the case where the minimal identifier is at the last hop before the loop. In this case, the algorithm will reset after y hops, then store the minimal identifier, and will not detect loop before the next reset. Therefore, it will only report the loop after $y + x_n + 2L - 1 = y(1 + \beta) + 2L - 1 = (\beta + 1) \cdot X - O(1)$. □

LEMMA 7. *Any deterministic algorithm that stores a single identifier must in the worst case use at least*

$$\min \{4, (3 + 2/\beta)\} \cdot X - O(1)$$

hops before identifying a loop.

PROOF. Throughout the lemma, we assume that $B = 0$ and consider the value of L. We take cases on β.

If $\beta \leq 0.5$ then the algorithm is not guaranteed to find the loop (it will keep on resetting before that, e.g., for $L = x_1$).

When $0.5 < \beta < 1$, we consider $L = \lfloor 2y/3 \rfloor + 1$ and assume that the minimal identifier is the last switch in the loop. Therefore, after y hops the packet has not reached the minimum for the second time and its identifier is reset. Then, before reaching the minimum for the second time, we have a reset at $x_n + y < 2y$ hops. Specifically, this means that the cycle is detected only after $4L - 1 = 4X - O(1)$ hops.

Next, consider $1 \leq \beta < 2$ (i.e., in this case, we have $x_n < 2y$). Let $B = 0, L = y + 1$, and consider the case where the minimal identifier is reset after this y-hops (i.e., it is at the end of the loop). Since x_n, we will not complete two cycles before the next reset; therefore, the loop is detected after no fewer than $4L - 2 = 4X - O(1)$ hops.

Finally, let $\beta \geq 2$. Here, let $L = \lceil \beta y/2 \rceil + 1$ and consider the case where the minimal identifier is the y'th hop in the loop. The algorithm will reset the identifier after seeing the minimum for the first time. It will then complete an entire cycle before seeing it again, and just before reaching it for the third time, it will reset again (as we reach $y + x_n$ hops). Therefore, the algorithm will make at least $y + x_n + 2L - 1 = y(\beta + 1) + 2L - 1 = (3 + 2/\beta)L - O(1)$ hops. □

We now infer the correctness of Theorem 5.

PROOF. Using the two lemmas above, we have that the detection time is lower bounded by

$$\max \{\beta + 1, \min \{4, 3 + 2/\beta\}\} \cdot X - O(1)$$

hops. Taking the minimum over all real β values we get a lower bound of $(2 + \sqrt{3})X - O(1) > (3.73 - o(1))X$. □

B USING MULTIPLE HASHES

Given an integer parameter $c \in \mathbb{N}$, consider partitioning each phase into c *chunks*. Intuitively, we are going to store c times as many identifiers, but each will only be active in a $1/c$ fraction of the phase. Specifically, during phase p, chunk j will get the minimal identifier of hops $\lceil b^p/c \cdot (j - 1) \rceil, \ldots, \lceil b^p/c \cdot j \rceil - 1$. The algorithm still compares the current switch identifier to all of the stored IDs and reports a loop if it finds a match.

The analysis only needs to change slightly: Lemma 3 can now show that after about at most $B + (b - 1)B/c + 1$ hops we have an identifier in the loop. In turn, this gives that the overall number of hops reduces to at most

$$2L + \max \left\{ \frac{2bL - 1}{b - 1}, B + (b - 1)B/c + 1 \right\}.$$

As an example, if we are allowed to store several identifiers, we can set $c = 2$ and $b = 7$ for a detection after at most $4.33X$ hops at the worst case.

Next, if we allow randomization, we can also consider using $H \in \mathbb{N}$ *hash functions*. Specifically, we assign each switch s with H identifiers $\{h_i(s) \mid 1 \leq i \leq H\}$ using random independent hash functions $h_1, \ldots, h_H$. A packet now contains H IDs $m_1, \ldots, m_H$, one for each minimum obtained by $h_1, \ldots, h_H$. When reaching a switch s, we compute its hashes and check if any of them matches the ones on the packet (i.e., whether there exists $i \in \{1, \ldots, H\}$ for which $h_i(s) = m_i$), and if so report a loop. Otherwise, for all $i \in \{1, \ldots, H\}$ we set $m_i = \min(m_i, h_i(s))$ if we are in the middle of a phase, or $m_i = h_i(s)$ if the last phase has ended and a new phase begun. Intuitively, if the phase is enough to complete two cycles over the loop, we can get *some* switch on the loop with *some* minimal identifier faster than if we had a single identifier. This is because the expectation of the minimum among H uniform variables in $\{0, \ldots, L - 1\}$ has an expectation lower than $L/(H + 1)$. Similarly, if the phase covers the loop $1 + \alpha$ times for some $\alpha \in [0, 1)$, the chance that we will get a minimal identifier in the first αL hops increases from α to $1 - (1 - \alpha)^H$. It then takes another L hops to complete another cycle and report the loop.

[4] For simplicity, we assume that such exists, otherwise the result will hold up to a term that vanishes as the loop grows longer.

REFERENCES

[1] 1997. Packet Loss Impact on TCP Throughput in ESnet. (1997). http://fasterdata.es.net/network-tuning/tcp-issues-explained/packet-loss/.

[2] R. Ben Basat, G. Einziger, and B. Tayh. 2020. Cooperative Network-wide Flow Selection. In *International Conference on Network Protocols (ICNP)*. IEEE.

[3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[4] Pavel Benáček, Viktor. Puš, Hana Kubátová, and Tomáš Čejka. 2018. P4-To-VHDL: Automatic generation of high-speed input and output network blocks. *Microprocessors and Microsystems* 56 (2018), 22–33.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM Computer Communication Review*.

[6] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. 2019. PURR: A Primitive for Reconfigurable Fast Reroute. In *Conference on Emerging Networking Experiments And Technologies (CoNEXT)*. ACM.

[7] P4 Language Consortium. 2018. P4 Switch Behavioral Model. (Jan 2018). https://github.com/p4lang/behavioral-model.

[8] The P4 Language Consortium. 2018. $P4_{16}$ Language Specification, version 1.0.0. (Jan 2018). http://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf.

[9] N. G. Duffield and Matthias Grossglauser. 2001. Trajectory Sampling for Direct Traffic Observation. In *Transactions on Networking, Volume: 9, Issue: 3*. IEEE/ACM.

[10] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2004. The Case for Separating Routing from Routers. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*. ACM.

[11] The P4.org Applications Working Group. 2018. In-band Network Telemetry (INT) Dataplane Specification. (Jan 2018). https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf.

[12] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. 2015. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[13] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. 2014. I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[14] Urs Hengartner, Sue Moon, Richard Mortier, and Christophe Diot. 2002. Detection and Analysis of Routing Loops in Packet Traces. In *Workshop on Internet Measurment (IMW)*. ACM.

[15] Vimalkumar Jeyakumar, Mohammad Alizadeh, Yilong Geng, Changhoon Kim, and David Mazières. 2014. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[16] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Symposium on Operating Systems Principles (SOSP)*. ACM.

[17] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[18] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. FlowRadar: A Better NetFlow for Data Centers. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[19] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[20] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[21] Srinivas Narayana, Mina Tashmasbi Arashloo, Jennifer Rexford, and David Walker. 2016. Compiling Path Queries. In *Networked Systems Design and Implementation (NSDI)*. USENIX Association.

[22] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[23] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. 2015. DC.P4: Programming the Forwarding Plane of a Datacenter Switch. In *Symposium on Software Defined Networking Research (SOSR)*. ACM.

[24] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. 2001. Hash-based IP Traceback. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[25] Ashwin Sridharan, Sue B. Moon, and Christophe Diot. 2003. On the Correlation between Route Dynamics and Routing Loops. In *Internet Measurement Conference (IMC)*. ACM.

[26] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Mao Morely, Scott Shenker, and Ion Stoica. 2005. HLP: A Next Generation Inter-domain Routing Protocol. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[27] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with Pathdump. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association.

[28] James Hongyi Zeng and Peyman Kazemian. 2012. Mini-Stanford Backbone. (2012). https://reproducingnetworkresearch.wordpress.com/2012/07/11/atpg/.

[29] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y. Zhao, and Haitao Zheng. 2015. Packet-Level Telemetry in Large Datacenter Networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM.