MINOS: Regulating Router Dataplane Actions in Dynamic Runtime Environments

Lei Xu† Ke Xu† Meng Shen§ Kui Ren¶ Jingyuan Fan¶ Chaowen Guan¶ Wen-Long Chen‡
†Tsinghua National Laboratory for Information Science and Technology, Beijing, China
†Department of Computer Science and Technology, Tsinghua University, Beijing, China
§School of Computer Science & Technology, Beijing Institute of Technology Beijing, China
¶Department of Computer Science and Engineering, State University of New York at Buffalo, USA
‡College of Information Engineering of Capital Normal University, Beijing, China
l-xu12@mails.tsinghua.edu.cn,xuke@tsinghua.edu.cn,shenmengnetlab@gmail.com,
kuiren@buffalo.edu,jfan5@buffalo.edu,chaoweng@buffalo.edu,cwl@csnet1.cs.tsinghua.edu.cn

ABSTRACT

Programmable routers are emerging as a promising alternative which facilitates the deployment of new network technologies, for example, software-defined networking; meanwhile, theirs programmability and openness also bring risks of security vulnerabilities. Prior work has concentrated on code security and encryption to improve router action honesty. In this paper, we exploit the feasibility of regulating actions on run-time dataplanes by detecting unexpected packet processing operations, which finally provides an honest and backdoor-proof router to operators. The main challenge is to monitor and regulate the action of router dataplane in dynamic runtime environment. Hence we propose Minos, a framework to regulate router actions on dataplanes. Minos takes Action Identifier (AID) as input to perform lookups in a pre-defined white list called Regulated Action Table (RAT), and it finally verifies that the action is (ab)normal. In the end, Minos achieves a pair of irreconcilable goals for security, *i.e.*, costs and effectiveness. We implement and evaluate Minos on Click and DPDK, separately. And our evaluation results show that Minos captures mal-actions with 2 mega-byte spatial costs and no more than 9% performance loss in both Click and DPDK.

CCS CONCEPTS

• **Networks** → **Routers**; **Network security**;

KEYWORDS

Router Security, Router Actions, Minos

ACM TUR-C '17, May 12-14, 2017, Shanghai, China
 ISBN 978-1-4503-4873-7/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3063955.3063996

1 INTRODUCTION

The past decade has witnessed large strides that have been taken in bringing high capacity, performance and analytical models into router designing [2, 4, 20, 21]. Programmable routers are emerging as a promising alternative which facilitates the deployment of new network technologies [6].Those advances come at the cost of increasing risks of security vulnerabilities.

In the recent years, routers running software to implement packet processing functions are susceptible to attacks increasingly, leading to system crash and information leak [1, 3, 5, 15–18]. Typically, Cisco router was reported to have found backdoors on devices [12, 13], which becomes a major concern for governments and operators. Even though all the modules and the invokings of modules in routers have been confirmed to be normal, they may be leveraged to achieve unexpected actions. For example, hacker is able to obtain privilege to manipulate router operation or alter the router behavior by particular updating model [11].

A big challenge of router security is to design a both effective and efficient mechanism for the dataplane that consists of various of functions. Prior work has concentrated on code security and encryption to improve router action honesty. Dobrescu *et al.* proposed a static verification tool to check security vulnerabilities on software dataplane [24]. Kim *et al.* utilized source authentication and path validation to prevent routers from doing unexpected actions [14]. They focus on static code verification and encryption cooperation among network nodes.

This paper is to exploit the feasibility of regulating actions on dataplanes to detect unexpected packet processing operations[1]. An action that is permitted by the operator is referred to as a *normal action*, or otherwise *mal-action*. The main problems this paper has addressed are listed as follows:

How to identify an action in run-time dataplanes?

Given a white list of all normal actions, how to verify the normality of real-time actions both efficiently and effectively?

[1]A mal-action may be caused by many reasons, such as a malicious operation by an attacker or a bug in dataplane design. Instead of focusing on identifying the reasons of them, this paper targets on mal-action detection.

In real-time router dataplane, the more effectively the verification performs, the more performance it consumes. Balancing these two seemingly irreconcilable goals represents a fundamental challenge in dataplane security design. To improve efficiency, we propose to verify pairs of adjacent components instead of entire action which lead to a bitmap construction of the white list. However, this comes with a few trade-offs that there are mal-actions that may escape verification. As a result, we introduce weight and the mechanism of removing loops in an action. Finally, the proposals of design complete these two seemingly incompatible goals.

This paper proposes Minos, a security framework by which all the actions of a router, not only normal actions but also mal-actions, will be monitored and justified in runtime. Constructing this framework needs two steps: an identification mechanism to identify distinguishable router actions, and a verification mechanism to check if the router action is legal. These two aspects together warrant the honesty of router actions in the end.

The three main contributions of this paper are listed as follows.

- We design the framework of Minos and use a bit-map construction to verify actions on router dataplanes, providing a security mechanism easy to be implemented by the third party.
- We introduce weight and removing loop mechanisms to prevent mal-actions from escaping verificationn, and prove that these methods are effective.
- We implement Minos on researchable Click and commercialized DPDK, separately. Evaluations show that Minos captures all the mal-actions with 2 megabyte spatial costs and no more than 10% performance loss in both Click and DPDK.

The rest of this paper is organized as follows. Section 2 summarizes the related work in the field of router security. Section 3 provides the characteristics of components. After that, the adversary models are presented in Section 4. Section 5 details the architecture and mechanisms of Minos. Then practical implementation and experimental results are illustrated in Sections 6 and 7. Finally the paper is concluded in Section 8.

2 RELATED WORK

Router security is of significance due to its crucial role in the Internet. The efforts in the field have been gradually concentrated on two main directions. On the one hand, industry spearheaded the security technology by configuring profiles ahead of time or repairing vulnerabilities after the emergence of security issue [3, 10, 19]. These protection measures require manual work and are hard to be automated. On the other hand, researchers recently expended considerable effort to leverage encryption and verification mechanisms to guarantee honest action of routers [14, 24, 28].

Kim *et al.* concerned source authentication and path validation to safeguard user data and transport by encryption

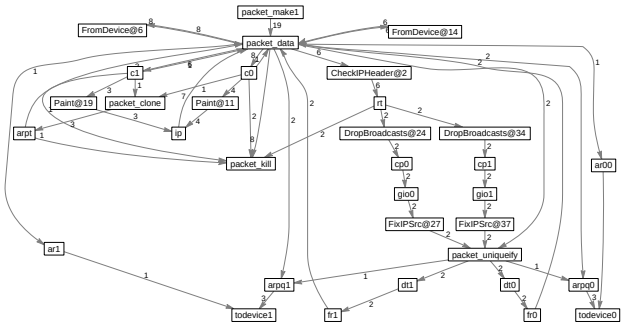


Figure 1: The invoking relationships of components on a typical router dataplane.

[14]. They leveraged a lightweight secure protocols to prevent router from doing malicious actions. They have used encryption to locate the root cause of and react to malicious actions. Dobrescu *et al.* provided an effective verification tool for software dataplane of router [24]. Their approach serves the purpose of checking exceptions and is a static analysis tool. These proposals are orthogonal and complementary to our approach which focuses on monitoring router actions in real time.

Control-Flow Integrity (CFI) is provided to prevent malicious transfer of control flow [25]. CFI is utilized to validate that a subtly targeted destination code is correct and harmless. The recent implementations [25–27] extended CFI technology onto commercial operating system kernels and GCC and LLVM. Minos in this paper provides another way to monitor control flow in dataplane.

3 COMPONENTS & ACTIONS

This section is the answer to the first question in Section 1, which is how to identify an action in run-time dataplane. We investigate components and actions of router dataplanes, and on the basis of observations we design CID and AID to represent components and actions.

Since this paper focuses on the security of software router platform that consists of various functions, the component in this paper has two properties. First, it is a packet-related processing unit that directly operates on packets. Second, it is a real-time processing unit on runtime router dataplanes. An action is considered as a pipeline of components in this paper.

3.1 Observations in Router Dataplane

This subsection investigates the components on the dataplane of Click router. We configure the Click router with two interfaces and basic routing functions. According to the declaration of component above, we view two types of processing unit as component. The first is the element whose member functions operate on packets. The second is all the member functions of Packet class.

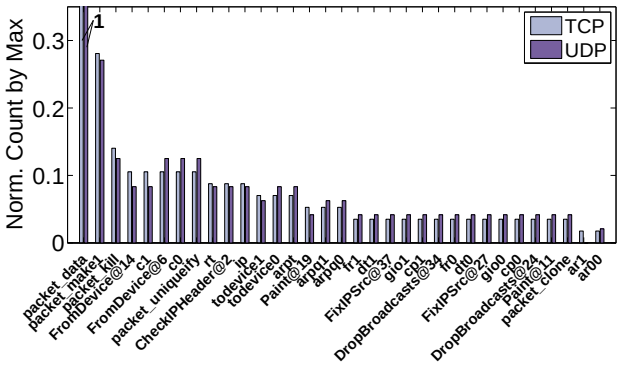
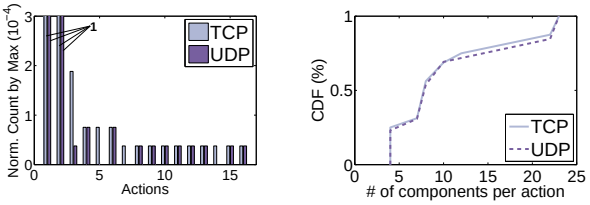


Figure 2: The invoking times of components among distinguishable actions.



(a) The invoking times of actions. (b) The number of involved components per action.

Figure 3: Observations of components and actions in the example.

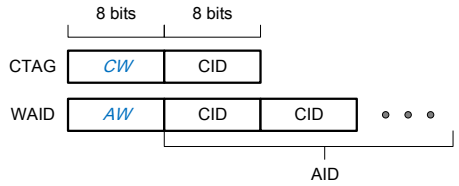


Figure 4: The formats of CTAG and WAID.

Before presenting observations, we list four conclusive results. First, the last component of router action is either SendPacket or KillPacket. Second, a router dataplane in real time does not contain a large number of components. Third, the set of regulated router actions is small comparing with the complete permutation set of all components. Fourth, the number of components within an AID is small. The following explores three perspectives from the perspective of the relationships between components, the characteristics of component and actions.

1) The Invoking Relationships of Components: This subsection describes the relationships of components among *distinguishable* actions on real time dataplanes. The action that has different components or different orders of components with other actions is viewed as a distinguishable action. For example, A-A-B and A-B are distinguishable from each other. The relationship topology in the mix scenario (TCP and UDP) is plotted in Fig. 1. In this figure, the node plotted by rectangle is the invoked component, the unidirectional edge represents the invoking relationship where the caller arrows toward callee. The number on the edge indicates the number of distinguishable actions using the edge. Although the figure is deduced from two-port IP router, the conclusion of multi-port router is still similar to that in Figure. 1, because the actions of each port are similar. The relationship topology is not too much complex and there are loops within. Some components is invoked frequently while others is invoked rarely.

2) The Characteristics of Components: This subsection analyzes the invoking times of components among distinguishable actions. As shown in Fig. 2, the invoking time of each component is normalized by the maximum count - the invoking time of packet_data. The x-axis shows the names of the invoked components. The light-purple and purple bars represent the times in TCP and UDP scenarios. The conclusion is that the frequently-invoked components account for a small minority. Although more and more components are designed, the realtime components on the running router dataplane are still the minority.

3) The Characteristics of Actions: This subsection provides the characteristics of distinguishable actions on router dataplanes, including the invoking times and the number of components, as shown in Fig. 3. To begin with, the numbers of distinguishable actions are 16 and 13 for TCP and UDP scenarios, respectively, which is a small set of actions. Meanwhile, 13 distinguishable actions are the same in both TCP and UDP scenarios. In summary, the invoking times of actions are directly related to upper applications, and the numbers of distinguishable actions and components within an action are small.

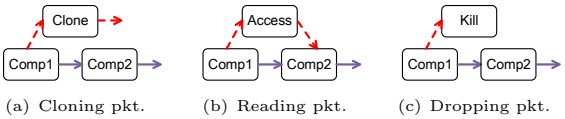
3.2 Design of CTAG and WAID

We propose Component Identifier (CID) to represent component and Action Identifier (AID) to represent action. We use binary number as CID, as it costs as small space and time as possible to be identified in real time. Based on the above observations, we use 8 bits of binary number for CID, Component Weight (CW) and Action Weight (AW).

Component TAG (CTAG) and AID with Weight (W-AID) is shown in Fig. 4. AID is the the CID sequence in the order of performing within action. AW is the sum of CWs in an action. The use of CID and AID is to achieve efficiency in checking router actions while the use of CW and AW is to achieve effectiveness to prevent mal-actions from escaping checking. The details of CW and AW are deferred to SubSection 5.4.1.

4 ADVERSARY MODELS

We aim at defeating mal-actions that are not permitted by operators. This section provides three typical mal-action



(a) Cloning pkt. (b) Reading pkt. (c) Dropping pkt.

Figure 5: Typical examples of mal-actions.

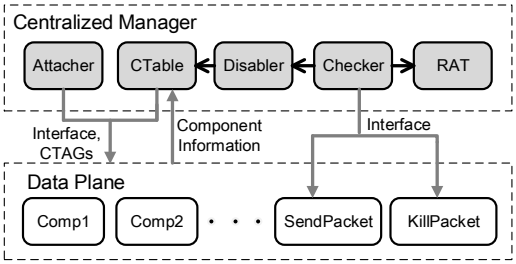


Figure 6: The construction of Minos.

examples in Fig. 5 to illustrate their features in dataplanes. The three examples are also similar to the adversary models in [14]. In Fig. 5, the components are plotted by rectangle and the invoking relationships are plotted by arrows.

In Fig. 5(a), the normal action is shown in blue-violet solid arrows while the mal-action of copying packet is shown in red dotted arrows. The feature of these mal-actions is that it is parallel to the corresponding normal action and thus leads to information leakage. As shown in Fig. 5(b), the Access component is invoked unexpectedly, and then the control flow returns to the expected component, *i.e.*, Comp2, leading to information leak or even altering payload. As shown in Fig. 5(c), after invoking Kill component, the mal-action is finished, leading to interruptive communications.

The above three representative mal-actions are probably utilized by adversaries, causing information leak, revenue loss and even state conflicts. In this paper, Minos is designed to defeat all the mal-actions.

5 DESIGN OF MINOS

In this section, we will detail the construction, the verification and the initialization of Minos, giving the answers to the second question in Section 1.

5.1 The Construction of Minos

The core of Minos is Centralized Manager (CM) which consists of five parts. This subsection briefly introduces the function of each part.

Figure 6 shows the five parts contained in CM. Attacher is a function that attaches CTAGs to WAID in temporary packet header and CTable stores all the CTAGs, both of which will be discussed in Section 5.2. Regulated Action Table (RAT) shown in Section 5.3.2 is a bit map storing all the normal WAIDs. Checker is a function used to check the correctness of WAID in RAT, whose details will be given out

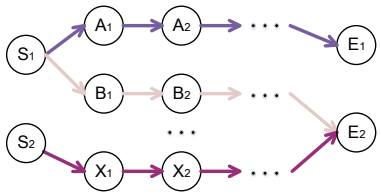


Figure 7: The normal AIDs without intersections except the first and last CIDs of AIDs and without loop within any AID.

in Sections 5.3 and 5.4. Lastly, Disabler is used to deal with mal-actions, which is presented in Section 5.5.

The following subsections presents the details of these mechanisms in Minos.

5.2 Attacher and CTable

This subsection explains the roles of Attacher and CTable. Once invoked, a component will attach its 8-bits CID to the tail of WAID in the temporary header of the packet, which is completed by Attacher. In addition, Attacher has to run another two tasks. One is to add 8-bits CW or EW to AW of WAID. The other is to check if the same CIDs exist in a WAID. If so, Attacher will increase the CID of the current component by 1 until no such CID exists in the WAID. Then it attaches the increased CID to WAID instead of the original CID.

The CTable is used to assign CWs and CIDs to components. The CWs and CIDs are calculated by the algorithm from normal AIDs, which will be described in Section 5.6. When the components submit their information, the CTable will assign corresponding CTAGs to the components.

5.3 The Efficiency of Minos

Router dataplane is sensitive to performance, hence we consider the efficiency of Minos as the first priority factor. This section describes the two mechanisms - Checker and RAT - that help improve performance of Minos.

5.3.1 Basis Idea of Verification. The Checker is used to verify whether the coming action or AID is normal. An AID implies the invoking relationships between its CIDs. In Minos, Checker reverts to checking *each pair of adjacent CIDs* within AID instead of the entire AID. Hence, the verification is simplified as checking the relationship between adjacent CIDs in an action.

Checking two adjacent CIDs rather than AIDs that have variable lengths offers two advantages. First, a pair of CIDs has fixed and shorter length than AID, simplifying the search algorithm. Second, two-CID entry makes the construction of RAT more regular.

Checking pairs of adjacent CIDs is effective only when all the AIDs satisfy two conditions: no intersection with each other and no loop within any AID. The following lemma provides this conclusion formally.

Definition 5.3.1. *AID is a directed path where node is CID and directed edge is the invoking relationship between CIDs. The set Normal whose elements are normal AIDs corresponds to a directed graph $G\langle N, E\rangle$ where N denotes CIDs and E denotes relationships between CIDs. The elements of set RAT are all pairs of adjacent CIDs of Normal AIDs. Use the elements of RAT to construct new AIDs whose first and last CIDs are the same as the first and last CIDs of AIDs in Normal. All the new AIDs form set Check.*

Lemma 5.1. *If all the AIDs in G has no intersections with each other except the first and last CIDs of AIDs and no loop within any AID, Check = Normal.*

Proof. Without loss of generality, use the graph in Fig. 7 as G of *Normal*, where S_1 and S_2 are the first CIDs while E_1 and E_2 are the last CIDs of G. The AIDs in G have no intersections with each other except the first and last CIDs and no loop within any AID. According to the constructing process of new AIDs from RAT, we can deduce that $Normal \subseteq Check$. Now we prove $Normal \supseteq Check$. If there is an AID α in *Check* which does not belong to *Normal*. Besides, assume α has the first CID S_1 and the last CID E_2. According to the constructing process of *Check*, α is constructed by the pairs of adjacent CIDs of AIDs in G. G has no intersections except the first and last CIDs of AIDs and no loop within any AID. Then α is $S_1B_1B_2...E_1$ which belongs to *Normal*, the assumption does not hold, $Normal \supseteq Check$. In conclusion, $Normal = Check$. □

In Section 5.4, the solutions are provided when the above conditions are released.

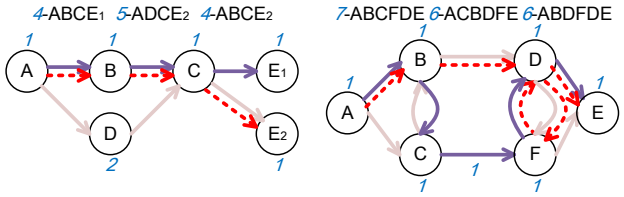
5.3.2 The Construction of RAT. As described above, the target of verification is two adjacent CIDs, leading to the 16-bit entry of RAT. RAT uses bitmap as its storage and search construction. In addition, 16-bit leads to 2^{16} entries of bitmap, *i.e.*, 8 Kilo Bytes (KB), an acceptable size to be deployed.

In RAT, the address of entry is the catenation of two CIDs. And the value 1 of entry indicates a normal relationship between the two CIDs while the value 0 means a mal-relationship. RAT is only readable upon the router's running, which prevents attacker from alternating its content. The content of RAT is calculated from normal WAIDs by Algorithm 1 described in Section 5.6.

5.4 The Effectiveness of Minos

As shown in Fig. 7, the efficiency of Minos is based on two conditions. Once the two conditions are released, the effectiveness of Minos will decrease. This subsection presents the solutions to this challenge.

5.4.1 The Component and Edge Weight. When normal AIDs have intersections, the mal-AID that is constructed by the pairs of normal adjacent CIDs exists. Figure 8(a) gives an example. Two normal AIDs, $ABCE_1$ (plotted by blue-violet arrow) and $ADCE_2$ (plotted by pink arrow), are



(a) The two WAIDs that have unshared CIDs. (b) The two WAIDs that share all CIDs with each other.

Figure 8: Examples of Component Weight (CW) and Edge Weight (EW).

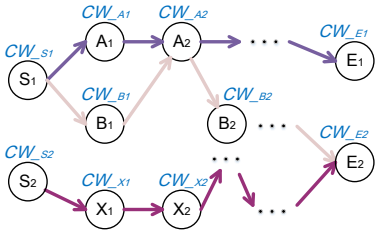


Figure 9: The normal WAIDs that have no conflicting AIDs and no loop within any AID.

presented, where A, B, C, D, E_1, E_2 are CIDs. According to the method in Section 5.3.1, all the pairs of adjacent CIDs, *i.e.*, AB, BC, CE_1, AD, DC and CE_2, are recorded in RAT. If both mal-AIDs $ABCE_2$ (plotted by red dotted arrow) and $ADCE_1$ (not plotted for clarity) are checked in RAT, they will be regarded as normal AIDs for that their all the pairs of adjacent CIDs exist in RAT.

To crack this hard nut, we provide components and actions with Component Weight (CW) and Action Weight (AW), respectively, where CW is a number and AW is the sum of CWs in an action. The AID with AW is called as WAID. If two actions have the same AW and they have intersections with each other except the first and last CIDs, they are called *conflicting* actions or *conflicting* AIDs. Finally, the problem is to identify and remove conflicting AIDs. The following Lemma describes the features of CW and AW.

Lemma 5.2. *Let CW be the number for CID and AW be the sum of CWs in AID. i) An AID has only one AW. ii) If two AIDs have at least one unshared CID, they can have different AWs by adjusting the CW of the unshared CID.*

Proof. Lemma 5.2 is evident and the proof of it is skipped. □

According to Lemma 5.2, in order to remove conflicting AIDs, we need to adjust the CWs of unshared CIDs of the conflicting AIDs to make the AWs of these conflicting AIDs different. We add CW by 1 to adjust CW. For instance, in Fig. 8(a), B, D, E_1 and E_2 are unshared CIDs. By adding 1 to CW of D to 2, the AWs of AIDs $ABCE_1$ and $ADCE_2$ are

4 and 5, respectively. Hence they are no longer conflicting AIDs.

In addition to checking pairs of adjacent CIDs for AID, AW is verified as well. And the entry of RAT is a triplet of AW-CID-CID, leading to the size of 2^{24} bits, *i.e.*, 2 Mega Bytes (MB), still acceptable to be deployed. The algorithm to assign suitable CWs to CIDs to remove conflicting AIDs is deferred to Algorithm 1 in SubSection 5.6. The following lemma proves the effectiveness of AW mechanism during verification.

Definition 5.4.1. *WAID is a directed path where node has CID and CW and directed edge is the invoking relationship between CIDs. The set $Normal_W$ whose elements are normal WAIDs corresponds to a directed graph $G_W\langle N, E\rangle$ where N is CIDs with CWs while E is the relationships between CIDs. The entry format of RAT is triplet AW-CID-CID and elements of RAT are from normal WAIDs of $Normal_W$. Use the elements of RAT to construct new WAIDs whose first and last CIDs are the same as the first and last CIDs of WAIDs of $Normal_W$. All the new WAIDs form set $Check_W$.*

Lemma 5.3. *If all the WAIDs in G_W have no conflicting AIDs and no loop within any AID, then $Check_W = Normal_W$.*

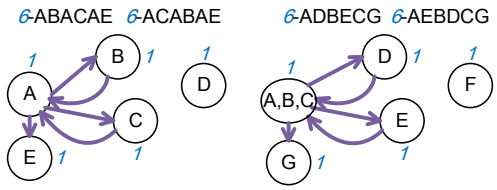
Proof. The proving process is similar to that of Lemma 5.1. Use the graph in Fig. 9 as G_W of $Normal_W$, where S_1 and S_2 are the first CIDs while E_1 and E_2 are the last CIDs of G. The WAIDs in G_W have no conflicting AIDs and no loop within any AID. According to the constructing process of new AIDs from RAT, we can deduce that $Normal_W \subseteq Check_W$. Now we prove $Normal_W \supseteq Check_W$.

Assume there is an AID α_W in $Check_W$ which does not belong to $Normal_W$. Besides, α_W has the first CID S_1 and the last CID E_2. So α_W consists of two parts. The first part is from S_1 to A_2 which belongs to the WAID $S_1, A_1, ..., E_1$ called as $WAID_1$. The second part is from A_2 to $E_@$ which belongs to the WAID $S_1, B_1, ..., E_2$ called as $WAID_2$.

Meanwhile, G_W has no conflicting AIDs and no loop within any AID. So $WAID_1$ and $WAID_2$ have different AWs. Then α_W has two AWs, which conflicts to Lemma 5.2. The assumption does not hold, $Normal_W \supseteq Check_W$. In conclusion, $Normal_W = Check_W$. □

Lemma 5.3 requires that normal WAIDs have no conflicting AIDs. This condition is completed by adjusting CWs of unshared CIDs, like B, D, E_1 and E_2 in Fig. 8(a). When there are no unshared CIDs between conflicting AIDs, the conflicting AIDs cannot be removed by differentiating AWs. As shown in Fig. 8(b), the conflicting AIDs $ABCFDE$ and $ACBDFE$ that share all the CIDs with each other always have the same AW, *i.e.*, 6. The mal-WAID 6-$ABDFDE$ will escape verification.

To solve this problem, we leverage unshared directed edges of conflicting AIDs. The unshared edges are assigned with Edge Weights (EWs) and the AW is the sum of CWs and EWs within an AID. The example is shown in Fig. 8(b), and



(a) A WAID with loops. (b) A WAID without loops.

Figure 10: Example of removing loops.

the directed edge CF is assigned with EW 1 and the AW of $ABCFDE$ is 7, eliminating the conflicting AIDs. AW and EW are calculated by Algorithm 1 in SubSection 5.6.

Thus far, CW and EW are introduced to eliminate conflicting AIDs. Another condition in Fig. 7 and 9 is that there is no any loop within AIDs. This condition will be further discussed in Section 5.4.2.

5.4.2 The Loops in Normal AIDs. When an action has more than one loops, mal-actions based on normal AW-CID-CID of RAT may escape the verification. A typical example is shown in Fig. 10(a), where a normal WAID 6-$ABACAE$ is plotted by blueviolet arrows and another CID D represents the CIDs belonging to other WAIDs. In this case, a mal-WAID 6-$ACABAE$ will escape verification, because its AW and all the pairs of adjacent CIDs can be derived from normal WAID.

To address this problem, we resort to removing the loops of AIDs. The approach is to treat the repeated CIDs in an AID as different CIDs. The following describes the details of the approach. First, when the Attacher attaches the current CID to AID, it checks all the previous CIDs in the AID if there are the same CID with the current CID. Once Attacher finds the same CID, it will add one to current CID. Second, the current CID is attached to the tail of the AID. Last, the CW of the current component is added to AW. Thus far, the loops in an AID are removed. The AID with removing loops is called RAID.

For instance, in Fig. 10(b), after removing loops for WAID 6-$ABACAE$, the WAID with Removing loops (WRAID) is finally 6-$ADBECG$. The process of finding repeated CIDs in AIDs will be discussed in Section 5.6. The following lemma reveals the fact that an RAID corresponds to only one AID.

Definition 5.4.2. *For function $f(\alpha) = \beta$, $\alpha \in AID$ and $\beta \in RAID$, both AID and $RAID$ are set of CID sequences. When α has repeated CIDs, e.g., $..C_1..C_1..C_1..$, f will increase the same repeated CID by an increment which is initialized with one and added by one when encountering the repeated CID, e.g., C_1. The other CIDs in α remains unchanged, finally leading to β, e.g., $..C_{10}..C_{11}..C_{12}..$ where $C_{1j} = C_1 + j$, $0 \le j \le T$ and T is repeated times of C_1. The repeated CID and corresponding added CIDs construct set $\mathbf{C_i}$, $0 \le i \le N$, $\mathbf{C_0} = \emptyset$ and N is the number of different CIDs in α, e.g., C_{10}, C_{11} and $C_{12} \in \mathbf{C_1}$.*

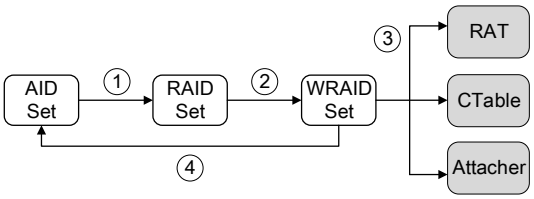


Figure 11: The process of initializing RAT.

LEMMA 5.4. *If* $\mathbf{C_i} \cap \mathbf{C_j} = \emptyset$ *where* $i \neq j$*, then* f *has an inverse.*

PROOF. A function has an inverse if and only if it is bijective, i.e., injective and surjective. For the injection, different CIDs, different orders and different number of repeated CIDs between elements in AID will lead to different elements of $RAID$ by f, because $\mathbf{C_i} \cap \mathbf{C_j} = \emptyset$. For the surjection, because $\mathbf{C_i} \cap \mathbf{C_j} = \emptyset$ for $i \neq j$, $C_i \neq C_j$. Hence different RAIDs correspond to different AIDs. □

After verification, the failed search of AW-CID-CID in RAT will trigger Disabler. The following subsection presents the process in Disabler.

5.5 The Disabling Process

Here we show the work progress of Disabler triggered by the failed verification of RAT. As discussed in Section 3.1, a packet in a dataplane has two destinies, *i.e.*, either to be sent out of the dataplane or to be freed. Preventing mal-actions from happening is to prevent the packet from being sent out or freed. Based on the analysis above, disabling mechanisms are three steps.

First, to prevent mal-action from sending packet out of dataplane, Disabler is installed in SendPacket component. When the action is verified as normal action, its packet is sent out, otherwise its packet is freed. Second, KillPacket component does nothing for mal-action. Third, Disabler is configured with an alarm for dataplane, which can be audio or visual alarm, to inform the operators of security issue once a mal-action is found. Meanwhile Disabler writes all the mal-action records into a log file for further diagnosis.

5.6 Initializing RAT

To initialize RAT, three jobs is needed, *i.e.*, obtaining the normal AIDs from router dataplanes, using normal AID set to initialize RAT, CIDs, CWs and EWs, and updating RAT if there are new normal WRAIDs.

The first job is as the following. At first, RAT is empty and CIDs are the accumulated number. In this case, all the actions will be viewed as mal-AIDs and recorded into log files by Disabler. The operator assumes responsibility of checking the AIDs in the log files if they are normal.

The second job is shown in Fig. 11. First, use AIDs to generate RAIDs, where the repeated CIDs within AID are accumulated according to method in Subsection 5.4.2. Second, based on the RAID set, we use an algorithm to assign CWs and EWs, generating WRAIDs, as shown in Algorithm 1. The algorithm is a graph-iterating algorithm called Allocate Weight Algorithm (AWA), which is discussed in the latter context. Lastly, RAT, CTable and Attacher are configured by the WRAID before running routers.

AWA aims at calculating CWs for CIDs and necessary EWs for conflicting AIDs. According to Subsection 5.4.1, AWA is to iterate all the nodes that have out-degree larger than one, and differentiate the AWs of *conflicting* edges that start from these nodes, as shown in the follow.

Algorithm 1 Allocate Weight Algorithm (AWA)

Require: *The directed graph G of normal RAIDs.*
1: *Initialize all the CWs to* 1*s and EWs to* 0*s*;
2: *ReInitialize CWs and EWs of loops*;
3: $conflict = get\_conflict\_action(G, \&act1, \&act2)$;
4: **while** $conflict == true$ **do**
5: $unshared\_cid = get\_unshared\_cid(act1, act2)$;
6: **if** *unshared_cid* **then**
7: $increase\_weight(unshared\_cid)$;
8: **else**
9: *unshared_edge*
10: $= get\_unshared\_edge(act1, act2)$;
11: $increase\_weight(unshared\_edge)$;
12: **end if**
13: $conflict$
14: $= get\_conflict\_action(G, \&act1, \&act2)$;
15: **end while**

The AWA is used before deploying Minos. In AWA, $get\_conflict\_action$ is designed carefully by optimizing its iterating methods to avoid dead loop. AWA's time complexity is $O(n^5)$ for that the $get\_conflict\_action$ is $O(n^3)$ and the $increase\_weight$ is $O(n^2)$.

The last job of updating RAT is shown in the forth step in Fig. 11. Once a new normal WRAID is provided by the operator, the new WRAID is used to generate new possible AID by recovering repeated CIDs and removing AWs. The operator has to check if the new AIDs are normal and update normal AIDs. The following steps are the same with the second job in the above.

6 THE IMPLEMENTATION OF MINOS

This section describes the implementations of Minos on Click and DPDK, separately [22, 23].

6.1 The Click Implementation

As described in Section 3, component refers to packet-related function. First, we chose in Click all the elements whose member functions operate directly on pa-ckets. It is easy to set Attacher in the *Push* member function of element. Second, other functions, such as packet-clone function, are added with the Attacher in the end of them. Attacher is realized as member function of the Packet Class.

Table 1: The effectiveness to 3 mal-actions.

Invoked Times	TCP		UDP		Mix	
	Mal	Normal	Mal	Normal	Mal	Normal
Copying pkt (5 MB)	To1 4888 To2 4891 Oth 6	Nt1 4888 Nt2 4891 Noth 14	To1 4888 To2 4886 Oth 4	Nt1 4888 Nt2 4886 Noth 12	To1 9777 To2 9778 Oth 4	Nt1 9777 Nt2 9778 Noth 10
Reading pkt (5 MB)	Oth 6	Noth 6	Oth 3	Noth 5	Oth 6	Noth 6
Dropping pkt (5 MB, $p = 0.5$)	Oth 6	Noth 7	Oth 3	Noth 3	Oth 6	Noth 8
Dropping pkt (60 KB, $p = 0.1$)	To1 8 To2 7	Nt1 70,Nt2 68 Noth 15	To1 1	Nt1 4,Nt2 4 Noth 31	To1 10 To2 10	Nt1 73, Nt2 77 Noth 18

To1/2:Todevice1/2 action, Nt1/2:Normal Todevice1/2 action, (N)Oth:(Normal) Other action.

CM is added in Router Class, where CTable and RAT are initialized in its construction function. CM is responsible to initialize CTable and RAT by configuration files whose content is generated by AWA. The Kill-packet function is wrapped by KillPacket as member function of element while the Send-packet functions in ToDevice Class are viewed as SendPacket. These two types of components are added with Checker in the end of them. The source code of Minos on Click is available online at https://github.com/minos2git/minos_click.git.

6.2 The DPDK Implementation

In DPDK, CM is deployed in rte_mbuf files that realizes many basic packet-related operations for the function library. The SendPacket is typical sending function, such as rte_eth_tx_burst in rte_ether.h and the KillPacket is a typical free function such as rte_pktmbu-f_free in rte_mbuf.h. Some macros that are used to read packet are replaced with functions as components. The configuration files of RAT and CTable generated by AWA are included by rte_mbuf files. All the packet-related functions are viewed as components. The lines of source code of Minos in DPDK are no more than one thousand.

7 EVALUATION

This section presents evaluations of Minos in terms of effectiveness and performance. The Click with Minos is deployed on 64-bit Ubuntu 12.04 with Intel Core i7-3770 CPU × 8, 8 GB DDR3 and two 100 Mega-bit-per-second (Mbps) RTL8139 ethernet adapters. The DPDK with Minos is deployed on 64-bit Ubuntu 14.04.3 with Intel Core i7-4790K CPU × 8, 32G DDR3 and four Intel Corporation I350-T4 Gigabit Network Connections.

The Click and DPDK router perform basic routing functions. In the experiment of Click, 5 MB and 60 KB data are transmitted in three scenarios, *i.e.*, TCP, UDP and mix scenarios. The DPDK is connected to Spirent TestCenter Packet Generator configured with TCP protocol and throughput test [8].

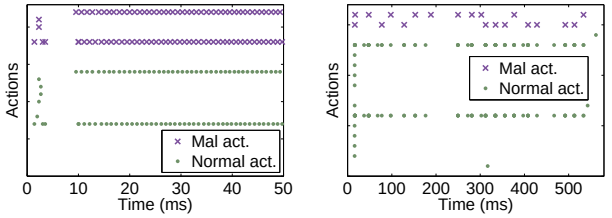
7.1 The Effectiveness of Minos

To investigate the effectiveness of Minos in Click, we added three mal-invokings in SendPacket component: cloning packet, reading packet and dropping packet randomly. All the actions that invoke SendPacket component will be changed into mal-actions if these three invokings occur. If Minos is not deployed, the precision and recall to detect them are around 60% [9]. Each mal-action experiment runs for 5 times, each result has a few differences because the lower layer protocols such as ARP in two hosts communicate with router, leading to extra actions being triggered. The distribution is the similar, so we chose one run to illustrate the effectiveness result in Table 1.

The number in Table 1 represents the captured times of each action by Minos. In the row of copying packet in Table 1, one host transmits 5-MB data to another host in three scenarios, TCP, UDP, and mix, separately. A normal action that invokes SendPacket will trigger a copying-packet mal-action. Hence in three scenarios, the numbers of mal-actions triggered by forwarding actions (To1/2) are equal to the numbers of normal forwarding actions (Nt1/2). The number of normal other actions (Noth) is larger than that of other mal-actions (Oth) in each scenario because there are normal actions that do not invoke SendPacket.

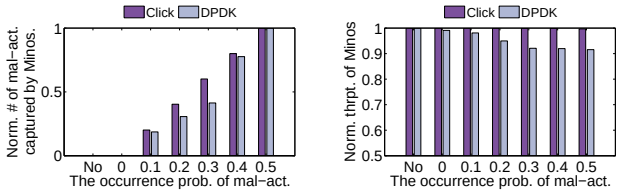
In the row of reading packet in Table 1, reading-packet mal-actions are conducted by invoking packet_d-ata component that involves in several loops in normal actions as shown in Fig. 1. We use this mal-action to examine the effectiveness of the weight and removing loops mechanisms. Take TCP scenario for example. The mal-action is "packet_make1, packet_data, ar1, todevice1, packet_data" where "todevice1, packet_data" is a mal-invoking. Minos increases the CID of packet_data to remove loops and confirms that this action is mal by its AW, revealing the weight and removing loops mechanisms are effective.

In the row of dropping packet with dropping probability 0.5, high to lead to mal-actions occurring once a packet arrives. Hence all the communications are prevented and the numbers of mal-actions are the same with the times in reading-packet row. In the row of dropping packet with dropping probability



(a) The copying action in TCP. (b) The dropping action with probability 0.1 in Mix.

Figure 12: The time points of actions.



(a) The number of mal-act. captured by Minos. (b) The throughput of Minos.

Figure 13: The performance of Minos with increasing mal-actions.

0.1. Minos captures all the dropping-packet mal-actions. TCP has transmitted 60 KB successfully in TCP and Mix due to its retransmission while UDP failed in transmission.

Figure 12 plots copying-packet and dropping-packet scenarios in run-time dataplanes, where y-axis represents actions while x-axis represents relative time.

Figure 12(a) plots a piece from 0 to 50 micro second (ms) where the two mal-actions are triggered by two forwarding actions. Because the mal-actions are parallel to normal actions, they do not impact normal actions, leading to the frequent occurrences of all actions. In Fig. 12(b), the density of actions is small because the mal-actions occur at the probability of 0.1 and they impact normal actions. In the mix scenario, TCP transmission completes successfully while UDP failed, leading to more types of normal actions than that of Fig. 12(a).

7.2 The Performance of Minos

This subsection investigates the performance of Minos for Click and DPDK. In terms of Click, we use iperf to measure the throughput [7]. In terms of DPDK, we use Spirent TestCenter Packet Generator configured with TCP protocol to measure the throughput [8]. For both Click and DPDK, the seven experiments are no Minos, Minos without mal-actions and Minos with mal-actions with probabilities 0.1 to 0.5. We get the normalized number of mal-actions and normalized throughput of Minos in Fig. 13.

As shown in Fig. 13(a), the normalized numbers of mal-actions captured by Minos are roughly proportional to the occurrence probability of mal-actions. The maximums of mal-actions in Click and DPDK are 55572 and 38400. This can be explained by the Minos features for Click and DPDK. Click has been deployed with Minos more completely leading to more components and actions than that of DPDK.

In Fig. 13(b), the normalized throughput decreases slightly with increasing numbers of mal-actions. The decreasing throughput is mainly due to the fact that Minos has to write mal-AIDs into files, which is I/O operation and slower than router bandwidth. The Click throughput is impacted less than DPDK because Click has 100 Mbps bandwidth that is impacted less than that of 1 Gbps of the DPDK router. The minimum/maxim-um throughputs of Click and DPDK are 92.0/93.1 Mbps and 860.9/940.2 Mbps, reflecting that Minos impacts the performance of router a little.

8 CONCLUSION

This paper proposes Minos to secure routers in runtime environment from the perspective of action. Based on the observations of actions in router dataplane and the features of mal-cation, we design Minos to identify all the active actions. To achieve efficiency, Minos adopts checking parts of action and a bitmap construction. To improve effectiveness, Minos leverages weight and removing-loops mechanisms. Minos is deployed on Click and DPDK, the evaluation reveals that the mechanisms have taken effect in runtime dataplane, satisfying the requirements of performance and costs.

The implementations of Minos reveal that it is a readily deployed security framework. It opens another door for securing router in the face of increasing backdoors in the current commercial routers. Minos is scalable by expanding RAT to large size and its construction makes it feasible to be deployed in SDN networks, which need our assiduous study in the future.

ACKNOWLEDGMENTS

This work was supported by the National Natural Foundation of China (61170292, 61472212), National Science and Technology Major Project of China (2015ZX0300-3004), the National High Technology Research and Development Program of China (863 Program) (2013AA-013302, 2015AA015601), EU Marie Curie Actions CR-OWN (FP7-PEOPLE-2013-IRSES-610524).

REFERENCES

[1] "Prism Project," https://nsa.gov1.info/dni/prism.html.
[2] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson and A. Greenberg, "Routing Design in Operational Networks: A Look from the Inside," in *SIGCOMM'04*.
[3] D. A. Maltz, J. Zhan, G. Xie, H. Zhang, G. Hjálmtýsson, A. Greenberg and J. Rexford, "Structure Preserving Anonymization of Router Configuration Data," in *SIGCOMM'04*.
[4] V. Paxson, "End-to-end routing behavior in the Internet," in *IEEE/ACM Trans. Networking'97*, 5(5):601-605.
[5] Y. Zhang and V. Paxson, "Detecting Backdoors," in *Pro. USENIX Security Symposium'00*.
[6] K. Xu, W. Chen, C. Lin, M. Xu, D. Ma and Y. Qu, "Towards Practical Reconfigurable Router - A Software Component Development

Approach," in *IEEE, Network, 2014*, 28(5):74-80.
[7] "Iperf," https://iperf.fr/.
[8] "Spirent packet generator," http://www.spirent.com/Products/TestCenter.
[9] A. Nandiz, A. Mandalz, S. Atreja, G. B. Dasguptaz and S. Bhattacharya, "Anomaly Detection Using Program Control Flow Graph Mining from Execution Logs," in *KDD 16*, August, 2016.
[10] "Cisco Security Products," http://www.cisco.com/c/en/us/products/security /product-listing.html.
[11] DEFCON Router Hacking Contest Reveals 15 Major Vulnerabilities, https://www.eff.org/deeplinks/2014/08/def-con-router-hacking-contest-success-fun-learning-and-profit-many.
[12] "Malicious Cisco router backdoor found on 79 more devices, 25 in the US," http://arstechnica.com/security/2015/09/ malicious-cisco-router-backdoor-found-on-79-more-devices-25-in-the-us/.
[13] "Cisco routers in at least 4 countries infected by highly stealthy backdoor," http://arstechnica.com/security/2015/09/ attackers-install-highly-stealthy-backdoors-in-cisco-routers/.
[14] T. H. Kim, C. Basescu, L. Jia, S. B. Lee, Y. Hu and A. Perrig, "Lightweight Source Authentication and Path Validation," in *SIGCOMM'14*.
[15] S. Sparks, S. Embleton and C. C. Zou, "A chipset level network backdoor: bypassing host-based Firewall & IDS," in *ASIACCS'09*.
[16] "Backdoor Found In Arcadyan-based Wi-Fi Routers," 2012. http://it.slashdot.org/story/12/04/26/1411229/ backdoor-found-in-arcadyan-based-wifi-routers.
[17] "RuggedCom - Backdoor Accounts in my SCADA network? You don't say," http://seclists.org/fulldisclosure/2012/Apr/277.
[18] A. Costin, J. Zaddach, A. Francillon and D. Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares," in *Pro. USENIX Security Symposium'14*.
[19] F. Le, G. G. Xie, D. Pei, J. Wang and H. Zhang, "Shedding Light on the Glue Logic of the Internet Routing Architecture," in *SIGCOMM'08*.
[20] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy, "Guarantee IP Lookup Performance with FIB Explosion," in *SIGCOMM'14*.
[21] G. Appenzeller, I. Keslassy and N. McKeown, "Sizing Router Buffers," in *SIGCOMM'04*.
[22] E. Kohler, R. Morris, B. Chen, J. Jannotti and M. F. Kaashoek, "The Click Modular Router," in *ACM Trans. Computer Systems'00*, 18(3):263-297.
[23] "DPDK Programmers Guide, Release 2.2.0," http://dpdk.org/doc/pdf-guides/prog_guide-2.2.pdf.
[24] M. Dobrescu and K. Argyraki, "Software Dataplane Verification," in *NSDI'14*.
[25] M. Abadi, M. Budiu, Ú. Erlingsson and J. Ligatti, "Control-Flow Integrity," in *CCS'05*.
[26] J. Criswell, N. Dautenhahn and V. Adve, "KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels," in *IEEE Symposium on Security and Privacy'14*.
[27] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano and G. Pike. Erlingsson, "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM," in *Pro. USENIX Security Symposium'14*.
[28] D. Naylor, M. K. Mukerjee and P. Steenkiste, "Balancing Accountability and Privacy in the Network," in *SIGCOMM'14*.