

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

WAVED: Principled Identification of Off-Path Exploitable Weak Verifications within the TCP/IP Protocol Suite

Yizhou Zhao and Xuewei Feng, *Tsinghua University;*
Min Li, *Zhongguancun Laboratory;* Ke Xu, *Tsinghua University*

https://www.usenix.org/conference/usenixsecurity26/presentation/zhao-yizhou

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

WAVED: Principled Identification of Off-Path Exploitable Weak Verifications within the TCP/IP Protocol Suite

Yizhou Zhao[1], Xuewei Feng[1]✉, Min Li[2], and Ke Xu[1,2]✉

[1]Tsinghua University, Beijing, China
[2]Zhongguancun Laboratory, Beijing, China
{zhaoyz24@mails.,xuke@}tsinghua.edu.cn, fengxw06@126.com, limin@mail.zgclab.edu.cn

Abstract

Off-path exploits targeting the fundamental TCP/IP protocol suite pose significant threats to the security of the Internet infrastructure. In particular, weak verifications of received payloads—arising from the lack of reliable information to validate or implementation flaws within the suite—lead to vulnerabilities that attackers can exploit to manipulate traffic, induce data loss, and disrupt services on victim servers. In this paper, we present the first systematic study of these vulnerabilities and introduce WAVED, a framework for identifying off-path exploitable weak verifications within the TCP/IP protocol suite implementation. At the core of WAVED, we develop a flow-, context-, and field-sensitive pointer analysis tailored to the TCP/IP kernel, and construct a Taint Propagation Graph (TPG) to model and trace data flow within the stack. By modeling byte-granularity taint propagation across diverse arithmetic operations, our approach can accurately locate specific input bytes associated with each constraint. Furthermore, direction-sensitive taint information is computed to accurately capture and differentiate the strength of constraints imposed by alternative branch outcomes, thereby significantly outperforming traditional byte-insensitive and direction-insensitive analyses. We evaluate WAVED on IPv4 and IPv6 across Linux 5.15, Linux 6.8, and FreeBSD 14.1. It precisely uncovers weak verifications leading to semantic vulnerabilities in TCP/IP and reveals 14 previously unknown vulnerabilities. We have responsibly disclosed these vulnerabilities to the affected OS vendors and have received acknowledgments from the Linux community.

1 Introduction

The TCP/IP protocol suite forms the foundation of modern Internet communication, enabling seamless data exchange across diverse networks worldwide [11]. Despite its critical importance, the protocol suite's long-standing design and inherent complexity have introduced subtle vulnerabilities [2, 3, 19]. In particular, one prominent class of vulnerabilities arises from weak verifications—inconsistencies or deficiencies in the verification of received payloads within the TCP/IP protocol suite. Off-path attackers can exploit these breaches to manipulate network routing [17] or launch Denial-of-Service (DoS) attacks [37]. Additionally, they can tamper with TCP/IP headers to hijack DNS service and TCP connections [15, 16, 30] or poison DNS and TCP traffic through IP fragmentation [4, 18, 43], leveraging weaknesses in the validation of forged ICMP error messages. The final steps of these attacks invariably trigger a risky operation (e.g., modifying critical shared variables). However, the reliable and verifiable exchange of information through such actions is often overlooked or cannot be fully guaranteed, as the lack of strong secret-based checks in stateless protocols and implementation-level logic oversights leave them insufficiently protected.

Due to the path explosion in the whole kernel protocol stack, it is impractical to identify these breaches through manual effort. Recently, a substantial body of work has focused on automating the security analysis of the kernel protocol stack [8, 9, 29, 42], but they all fall short of detecting weak verifications. Chen *et al.* [9] applied implicit taint analysis to packet-injection vulnerabilities, classifying each branch check as low- or high-entropy. However, they fail to measure constraint strength automatically except for simple comparisons like constants or equality, necessitating manual on-the-fly filtering that renders the approach semi-automated and unscalable. Cao *et al.* [8] leveraged model checking to detect violations of the non-interference property. However, over-abstraction can alter the actual constraints along each path, and the need for manual expertise renders this approach impractical for full IPv4/IPv6 kernel analysis. Yu *et al.* [42] and Man *et al.* [29] explicitly specify side-channel path patterns and automate their discovery. Nonetheless, side channels arise from two paths with differing attacker-observable behaviors—unlike weak verification (a single path causing a security violation), these works focus on only path existence rather than constraint strengths. Moreover, the symbolic execution employed in [29] is too computationally expensive and inefficient for full-stack analysis.

Based on these observations, in this paper, we conduct *the*

first systematic study dedicated to uncovering off-path exploitable weak verifications within the fundamental TCP/IP protocol suite. Our key insight is that *the strength of a path constraint is proportional to the knowledge an off-path attacker must possess to satisfy it*. We operationalize this metric as the set of *strongly constrained (bounded checked)* bytes in the packet forged by an attacker. To measure the strength of path constraint to each byte, we propose direction-sensitive taint information, modeling the direction of constraints inside every tainted value. Therefore, we introduce WAVED (WeAk Verification Exploitation Detector), a *novel, automated* framework that leverages our proposed path constraint metric to identify execution paths guarded only by weak verifications.

Specifically, WAVED explores all possible execution paths from the protocol stack's input to semantically risky operations, checking whether these paths are weakly constrained in control flow, which would make them exploitable by off-path attackers. At a high level, WAVED operates in four key steps. Firstly, we develop a flow-, context-, and field-sensitive pointer analysis tailoring to the TCP/IP kernel analysis, which precisely handles Get Element Pointer (`GEP`) instructions, thereby enabling accurate object resolution and parent-structure addressing, serving to construct the indirect value flows required for subsequent taint analysis. Second, we perform a Meet-Over-all-Paths (MOP) taint analysis that is byte-granularity, as well as flow-, context-, and field-sensitive. This analysis is built upon the results of the pointer analysis and scales efficiently to the entire IPv4/IPv6 kernel. In this phase, we construct a Taint Propagation Graph (TPG) to represent data flows between variables, explicitly modeling byte-level taint propagation across various arithmetic operations. Thirdly, we introduce direction-sensitive taint information (`branch impact`) to accurately model the strength of constraints at each branch transition through back resolution on the TPG, precisely distinguishing weak constraints from strong ones. This allows WAVED to explore all potentially risky paths while minimizing false positives caused by strong constraints. In this step, distinct branch transition paths within each function are deduplicated, summarized, and stored alongside their accumulated constraints as `function summary` for subsequent analysis. Finally, a Depth-First Search (DFS) is employed to traverse all feasible execution paths from the program entry, deduplicating and accumulating path constraints. Upon reaching a pre-labeled risky operation, if the accumulated constraint is identified as weak by our *strength filtering* mechanism, the path is deemed exploitable and reported to the user.

We implement WAVED based on the framework of LLVM [26] and the SVF (Static Value-Flow) analysis tool [39]. To comprehensively evaluate the effectiveness and generality of WAVED, we apply it to both the IPv4 and IPv6 implementations of three popular OS kernel versions, Linux 5.15, Linux 6.8 and FreeBSD 14.1, and discovered 19 weak-verification vulnerabilities (among them, 14 are new). These vulnerabilities can be exploited by off-path attackers to poison routing caches and inject crafted packets into victim servers, causing severe disruption. We have responsibly disclosed all identified vulnerabilities to the Linux and FreeBSD security teams. We further evaluate and demonstrate the precision and reduction of WAVED in handling direction-sensitive constraints, showing that it achieves negligible optimization in path discovery.

Contributions. Our main contributions are as follows:

- We propose a novel method that utilize direction-sensitive taint information to quantify the strength of path constraints, and conduct the first systematic study of weak-verification vulnerabilities in the fundamental TCP/IP suite.
- We design and implement WAVED, an automated tool that integrates pointer analysis, byte-level taint tracking, and direction-sensitive constraint measuring to efficiently detect off-path exploitable weak verifications in TCP/IP stacks. The tool provides precise modeling of execution paths, minimizes false positives, and is released at https://doi.org/10.5281/zenodo.17896120.
- We conduct a comprehensive evaluation on both IPv4 and IPv6 implementations of three popular OS kernel versions. WAVED reports 19 weak-verification vulnerabilities, including 14 previously unknown ones that we have validated in official OS releases. These vulnerabilities have been responsibly disclosed, with acknowledgments from Linux.

2 Background

In this section, we briefly introduce the background of how weak verifications lead to problems (e.g., shared variable modification) within the TCP/IP protocol suite, and illustrate how off-path attackers can exploit them.

2.1 Shared Variables within the TCP/IP Suite

Resource sharing across protocol interactions in the TCP/IP suite enables efficient communication and cohesive operation of protocols. Key information—including network parameters, state variables, and routing details—is exchanged to support cross-layer functionality. A notable example is the path MTU, maintained at the IP layer for a given destination and shared across all transport-layer sessions to this destination [31–33]. Any session can update it, potentially cross-affecting other sessions and introducing risks such as unintended fragmentation or traffic delays. Similarly, ICMP and TCP challenge ACK rate limits are globally applied, providing operational efficiency but creating potential security risks if manipulated [6, 7, 28].

The layered and distributed nature of the TCP/IP stack often prevents reliable verification of shared variables. Stateless protocols like UDP and ICMP exacerbate this issue, as

they lack session state and mechanisms to validate processed data. Consequently, attackers can exploit weak verifications in shared variables, causing traffic disruption or unauthorized resource access [17]. While essential for performance, shared variables must be carefully managed and verified to mitigate these security risks.

2.2 Illustrative Example

Figure 1 illustrates how an off-path attacker exploits weak verification during the update of a shared variable, specifically the path MTU maintained at the IP layer for all transport-layer sessions to the same destination. By manipulating this variable, the attacker can induce unintended IP fragmentation, enabling TCP traffic poisoning via forged IP fragments [18].

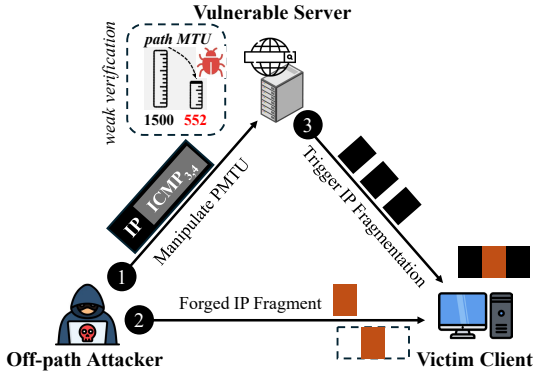


Figure 1: An illustrative example of a weak-verification vulnerability within the server's TCP/IP protocol suite in Linux.

To launch the attack, the off-path attacker spoofs the source IP of an intermediate router and sends a crafted ICMP `Destination Unreachable` message with the `Fragment Needed` code and the `DF` (Don't Fragment) bit set. Normally, such a message informs the server of the maximum transmissible packet size along a path to the client, as part of the Path MTU Discovery (PMTUD) mechanism [32, 33]. However, because any router could issue this message, a spoofing attacker can forge it to manipulate the server's path MTU state[1].

Specifically, the attacker embeds a crafted ICMP `Echo Reply` (Type=0, Code=0) into the forged ICMP message and sends it to the target server. Due to weak verification in the stateless ICMP handling, this packet propagates through the TCP/IP stack, updating the shared Path MTU variable and triggering cross-protocol interactions, potentially exposing vulnerabilities. In Linux systems, the forged ICMP message is sequentially processed by `ip_rcv()`, `ip_local_deliver()`, `icmp_rcv()`, and `icmp_unreach()`. The embedded echo reply is then handled by `icmp_err()` (or protocol-specific handlers like `__udp4_lib_err()` for UDP and `tcp_v4_err()` for TCP[2]). During this handling, the kernel calls `ipv4_update_pmtu()` to update the Path MTU maintained for the client, e.g., reducing it from 1500 to 552 octets.

Although RFCs [1, 34] specify that ICMP error messages should include at least 28 octets of the original packet for verification, the Linux kernel directly updates the Path MTU when the embedded packet is an ICMP `Echo Reply`, bypassing further checks (Listing 1). Unlike TCP, which uses unpredictable sequence numbers for validation, stateless protocols such as ICMP or UDP lack such identifiers, making precise verification infeasible and creating a semantic gap [17]. Consequently, the off-path attacker can manipulate the shared Path MTU via a forged ICMP `Fragment Needed` message containing a crafted `Echo Reply`, inducing unintended fragmentation and enabling TCP traffic poisoning via forged IP fragments.

```
......
if (icmph->type != ICMP_ECHOREPLY) {
    ping_err(skb, offset, info);
    return 0;
}

if (type == ICMP_DEST_UNREACH && code ==
    ICMP_FRAG_NEEDED)
    ipv4_update_pmtu(...);
```

Listing 1: A code snippet of verification logic for received ICMP-encapsulated ICMP error packets in Linux 6.8.

This lack of verification in execution paths towards sensitive operations remains pervasive in TCP/IP protocol stack implementations, creating abundant opportunities for exploitation. A critical distinction is that attackers can circumvent these checks with minimal prior knowledge—requiring no secrets, unlike robust checks in `tcp_v4_err()` where the attacker must know specific ports together with highly-secure sequence numbers. This raises a fundamental question: *How can we quantify the knowledge an attacker needs to bypass the checks along a given execution path?*

3 Threat Model & Overview

In this section, we present the threat model of off-path attacks that exploit weak-verification vulnerabilities in a host's TCP/IP protocol suite, our insight and the main challenges to uncover these vulnerabilities. Finally, we introduce the overview of our methodology, WAVED.

3.1 Threat Model

To investigate the most threatening scenario, our threat model demonstrates how an off-path attacker can exploit a weak-verification vulnerability to execute risky operations, as illustrated in Figure 2. Such effects can further compromise

[1] This example uses IPv4, but analogous attacks exist for IPv6 via ICMPv6 Packet Too Big messages [12]. Our framework also detects weak verifications in IPv6 implementations.

[2] Different embedded packet types trigger different handlers.

end-to-end communication between the victim server and client, ultimately enabling a cross-layer attack.

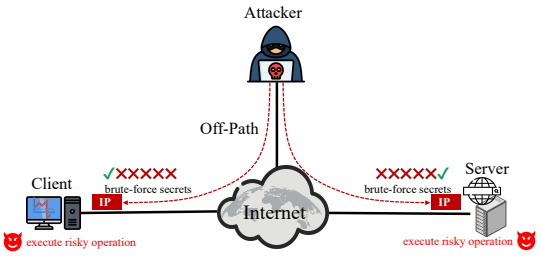


Figure 2: Threat model of WAVED.

Our threat model involves three entities: a server, a client, and a malicious off-path attacker. The server hosts various online services over protocols such as TCP, UDP, and ICMP (e.g., ping for reachability). The client communicates with the server via stateful connections (e.g., TCP) or stateless requests (e.g., UDP, ICMP), enabling normal client-server interactions. The client's and server's TCP/IP protocol suite implementations may contain weak-verification vulnerabilities that could be exploited to perform severe attacks. The off-path attacker, without direct access to the communication context, can spoof source IP addresses and send crafted packets to the client or server. By brute-forcing weak secrets used in critical branches within the TCP/IP stack, the attacker can bypass branch checks, perform risky operations (e.g., manipulate shared resources), directly abort the connection, or lay the groundwork for subsequent attacks.

In practice, attackers may leverage bulletproof hosting or rented machines to launch spoofing-based attacks [15,30], impersonating other hosts or routers to inject forged packets into the server. Despite being off-path, this attacker model poses a substantial threat: it requires only basic IP spoofing capabilities and is difficult to trace, yet can trigger state modifications or other exploitable conditions on the victim.

3.2 Insight: Modeling Constraint Strength with Direction

As illustrated in Section 3.1, since an off-path attacker is often restricted to blind packet injection, the attack difficulty is strictly bounded by the search space of the input values they must guess to satisfy the verification. To answer the question in Section 2.2, we propose quantifying the required knowledge as the number of input bytes they need to know.

However, every part of the input is checked in the processing, and *not all constrained bytes effectively contribute to the difficulty of an attack*. We observe that weak constraints, typically manifesting as unbounded inequalities or one-sided comparisons (e.g., pkt.pmtu < current_pmtu for PMTU updating, or out-of-window check for TCP sequence number), leave a vast portion of the input space valid, allowing attackers to easily construct satisfying packets by filling a considerably small or large value without precise knowledge. In contrast, strong constraints—bounded equality or range checks (e.g., in-window check for TCP sequence number)—drastically reduce the solution space and serve as secrets that sharply increase blind-injection cost. Consequently, the problem of modeling constraint strengths and detecting weak verifications transforms into *solving for the set of strongly constrained bytes along an execution path*.

Distinguishing strong constraints from weak ones necessitates identifying the constraint direction. However, traditional taint analysis (e.g., [9,27,42]) is limited to detecting the mere presence of data dependencies, failing to capture this nuance. Therefore, to effectively differentiate constraint strength, it is necessary to design a taint analysis scheme capable of identifying the direction of constraints—a property we term *direction-sensitivity*. Furthermore, this analysis must operate at *byte-granularity* to accurately quantify the exact number of strongly constrained bytes.

3.3 Challenges

Developing a systematic tool to calculate direction-sensitive taint information and analyze weak-verification vulnerabilities within the TCP/IP protocol suite presents several challenges:

Challenge I. *Precise pointer analysis in protocol-specific kernel code.* Accurate data-flow analysis requires resolving points-to sets with precision. Unlike user programs, protocol-specific kernel code introduces complex call hierarchies and sophisticated pointer arithmetic. Context-insensitive analysis conflates objects across calls (e.g., a `malloc()` wrapper in different contexts is considered as a single object), while complex pointer operations such as pointer subtraction (common in kernel code, e.g., `container_of()` in Linux) cannot be correctly handled by traditional field-index-based methods [27, 39]. This motivates our context-, flow-, and field-sensitive pointer analysis tailored to diverse kernel pointer operations.

Challenge II. *Efficiently capturing related constrained bytes in each branch check.* To precisely measure the path constraints, conventional object-granularity taint analysis does not work well, as it can only determine whether a branch condition is related with the attacker's input (i.e., crafted packets), but fails to identify which part of the input is constrained. To cope with this, a byte-granularity taint analysis is what we propose. However, propagating the taint information in each path individually cannot scale to large programs like the whole kernel protocol stack due to path explosion [5], therefore we design a Meet-Over-all-Path approach based on the pointer analysis results, which significantly accelerates this process.

Challenge III. *Precisely modeling branch constraints with direction-sensitive taint design.* As illustrated in Section 3.2,

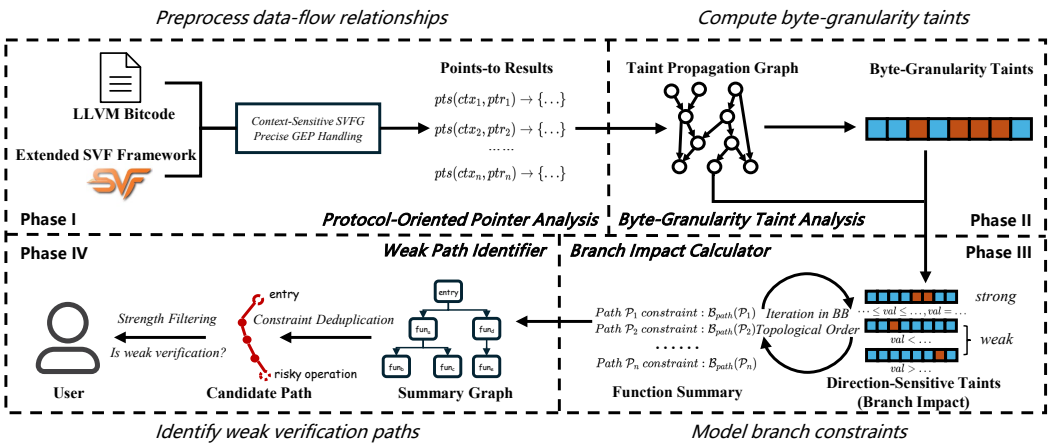


Figure 3: Overall framework of WAVED.

traditional taint analysis identifies only the existence of a dependency, failing to distinguish the *direction* of a constraint. Consequently, it cannot differentiate between strong constraints (e.g., strictly in-window) and weak ones (e.g., out-of-window). To capture this nuance, the taint analysis must be direction-sensitive. Furthermore, in practice, constraints from multiple branches merge along execution paths; thus, the analysis must support logical composition—specifically the conjunction and disjunction of direction-sensitive taints. Moreover, kernel implementations frequently encapsulate sub-constraint checks within helper functions, effectively transforming control-flow constraints into data-flow values (i.e., implicit taint). Therefore, a significant challenge lies in extending our analysis to incorporate such implicit propagation. Realizing these, we introduce a direction-sensitive taint calculation method to overcome these hurdles.

3.4 Overview of WAVED

Our approach, WAVED, searches for possible execution paths from the packet receiving entry to risky semantic operations, incrementally accumulating the constraints of control-dependent branch outcomes along each path. Based on this intuition, WAVED derives the exact path constraint that characterizes the currently explored control flow. It then employs a *strength filtering* strategy to the current path constraint-comparing the number of strongly constrained input bytes against a preset threshold while filtering out paths strongly constrained by highly-secure critical fields (e.g., TCP/DCCP seq/ack, SCTP vtag). In this way, WAVED can determine whether the path is guarded by weak verification, thus exposing risky operations which can be easily triggered by an off-path attacker.

Figure 3 illustrates how WAVED identifies potential exploitable weak verifications within the implementation of the TCP/IP protocol suite. First, WAVED performs a flow-, context-, and field-sensitive pointer analysis on the LLVM bitcode of the protocol stack, collecting implicit data flows under address-taken variables. Based on the results, it constructs a Taint Propagation Graph (TPG) that models byte-level data flows among variables, upon which a Meet-over-All-Paths (MOP) byte-granularity taint analysis is carried out. Next, direction-sensitive taints are derived from the analysis to accurately capture the strength of constraints. Finally, WAVED explores execution paths leading to risky operations, incrementally merging direction-sensitive constraints along each path, and reports those paths that are deemed weakly constrained. In this way, WAVED consists of four major components:

Protocol-Oriented Pointer Analysis (Section 4.1). A flow-, context-, and field-sensitive pointer analysis, leveraging byte-offset-based `GEP` handling to address Challenge I.

Byte-Granularity Taint Analysis (Section 4.2). A Meet-Over-all-Paths taint analysis based on the TPG, and achieves byte-granularity, addressing Challenge II.

Branch Impact Calculator (Section 4.3). A dedicated module designed to extract direction-sensitive taints from the taint analysis results and the TPG, enabling WAVED to effectively differentiate the impacts of alternative branch outcomes and quantify the strength of constraints, thereby addressing Challenge III. In addition, a `function summary` is computed during the resolution of branch constraints within each function, deduplicating paths with identical constraints and improving the efficiency of subsequent exploration.

Weak Path Identifier (Section 4.4). A Summary Graph, derived from the `function summaries`, is constructed and traversed to search for execution paths leading to risky operations. By evaluating the accumulated constraints along each path, weakly constrained ones are identified and reported.

4 Design Details

In this section, we present the detailed design of four main components in WAVED.

4.1 Protocol-Oriented Pointer Analysis

To address Challenge I, a flow-, context, and field-sensitive pointer analysis, orienting at kernel protocol stacks, is developed in WAVED. We introduce the *Protocol-Oriented Pointer Analysis* by extending the most precise whole program analysis in the SVF framework, Flow-Sensitive Whole Program Analysis [39]—a widely recognized, sound MOP solution for precise, scalable, flow- and field-sensitive pointer analysis.

The working procedure of a flow-sensitive pointer analysis in SVF can be summarized in three steps: (1) run an Andersen-style points-to analysis, (2) build a Sparse Value-Flow Graph (SVFG) based on the points-to results, and (3) propagate points-to information across the SVFG. Observing that the approach in SVF not only lacks context sensitivity to distinguish memory objects in different contexts, but also uses coarse-grained flattened field indices (the inconsistency of field sizes cause imprecision) to represent different field objects, leading to imprecision when handling `Casts` and negative-offset `GEPs` (frequently occurs in kernel code), to address the aforementioned limitations, we replace the original SVFG with a context-sensitive variant and enable byte-offset-based handling for `GEP` instructions, thereby improving precision in both inter-procedural context modeling and field-sensitive pointer computations.

Context-Sensitive SVFG. To achieve context sensitivity, we introduce *Context-Sensitive SVFG* (CSSVFG), a context-sensitive variant of SVFG. The context of each CSSVFG node is a sequence of at most 3 call sites (e.g., for the call chain $\cdots \to c_1 \to c_2 \to c_3$, the context is $\langle c_1, c_2, c_3 \rangle$). A call site refers to a pair of a call instruction and its target function, which is precomputed and indexed during the construction of the context-insensitive call graph in SVF. The construction of CSSVFG proceeds in three steps (detailed in Algorithm 1):

1) Context Collection. We first compute the set of possible contexts $\mathcal{C}(f)$ for each function f based on the context-insensitive call graph computed by FSWPA. The contexts are collected by traversing from the entry function.

2) Node Duplication. Each node in the original SVFG is duplicated for every context in $\mathcal{C}(f)$, producing $|\mathcal{C}(f)|$ context-sensitive versions. This enables CSSVFG to distinguish value flows across different calling contexts.

3) Edge Construction. We model both intra-context and inter-context edges in CSSVFG: Intra-context edges are copied from the original SVFG; Inter-context edges for direct calls are added during construction, and those for indirect calls are resolved later via the on-the-fly context-sensitive call graph. Only edges that comply with context composition rules are included.

Byte-Offset-Based `GEP` Handling. To accurately handle `GEP` operations, we replace the traditional field-index-based strategy with a byte-offset-based field analysis. For compatibility, we maintain a *per-type bidirectional mapping between byte offsets and field indices*. When computing pointer offsets, the field index is translated into a byte offset, added by the offset, and mapped back to a field index. This enables all `GEP` instructions to be modeled as concrete byte shifting, preserving their full semantics.

With these enhancements, the Protocol-Oriented Pointer Analysis achieves flow-, context-, and field-sensitivity, accurately modeling complex pointer arithmetic patterns common in kernel-level code. A context-sensitive call graph is also constructed on-the-fly during the analysis.

4.2 Byte-Granularity Taint Analysis

As noted in Challenge II, traditional object-granularity taint analysis methods [9, 27, 42] cannot accurately locate related bytes of a branch condition. Fortunately, with the results of our pointer analysis, a precise and byte-granularity taint analysis can be performed. We forgo path-sensitive analysis as it is computationally prohibitive due to path explosion [5]. Instead, we adopt the MOP formulation, achieving a necessary balance between kernel-scale scalability and rigorous flow sensitivity [36]. To implement this, WAVED first constructs a *Taint Propagation Graph* (TPG) to capture byte-granularity value-flow relationships between variables.

Taint Propagation Graph. Inspired by the design of CSSVFG, which represents object-granularity data flow information, TPG leverages a structure analogous to the CSSVFG, directly linking top-level variables using explicit data dependencies in the IR, and connecting address-taken variables with sparse value-flow edges relying on the points-to information provided by the Protocol-Oriented Pointer Analysis. According to our design, TPG is *context-sensitive*-the same statements in different contexts are modeled as different TPG nodes.

In order to represent byte-granularity data flows, each edge in the TPG is associated with *an embedded bitmap of up to 8×8 in size*, following the LP64 data model adopted by Unix-like systems (8 bytes) [13]. Each cell $map[i][j]$ (zero-indexed and little-endian) in a TPG edge indicates whether a taint on byte i of the source node can propagate to byte j of the destination node, and the taint propagation can be represented as a matrix-multiplication-like procedure, supposing the multiplication of $map_{u\to v}$ and $map_{v\to w}$ equals $map_{u\to w}$ (the taint propagation from node u to node w), then for $\forall i, j \in [0...7]$, we have:

$$map_{u\to w}[i][j] = \bigvee_{k=0}^{7} map_{u\to v}[i][k] \wedge map_{v\to w}[k][j] \quad (1)$$

with a total time complexity of $\Theta(n^3)$ ($n = 8$). To reduce space overhead and accelerate computation, the 8×8 taint propaga-

Type	Taint Bitmap	Corresponding Instructions/Relationships
LowToHigh()	$\forall 0 \le i \le j < 8:\ map[i][j] := 1$	*GEP(base pointer), Add, Sub, Mul*
Copy()	$\forall 0 \le i < 8:\ map[i][i] := 1$	*Cast, Phi, Store, Select, Branch, FNeg, And(non-constant), Or(non-constant), Xor, Call(param and return value), Indirect Value-Flow Edges(address-taken variables)*
LogicOff(offset)	$\forall \lceil \frac{offset}{8} \rceil \le j-i \le \lfloor \frac{offset+7}{8} \rfloor,\ 0 \le i,j < 8:$ $map[i][j] := 1$	*Shl(constant offset), LShr(constant offset)*
ArithOff(offset, byteSz)	$\forall \lceil \frac{offset}{8} \rceil \le i-j \le \lfloor \frac{offset+7}{8} \rfloor,\ 0 \le i,j < byteSz:$ $map[i][j] := 1$ $\forall 0 \le byteSz-1-i \le \lceil \frac{offset+7}{8} \rceil:\ map[byteSz-1][i] := 1$	*AShr(constant offset)*
And(const)	$\forall const\ \ \&\ \ (1 \ll i) \neq 0,\ 0 \le i < 8:\ map[i][i] := 1$	*And(constant operand)*
Or(const)	$\forall const\ \ \&\ \ (1 \ll i) = 0,\ 0 \le i < 8:\ map[i][i] := 1$	*Or(constant operand)*
Full()	$\forall 0 \le i,j < 8:\ map[i][j] := 1$	*All other kinds*

Table 1: Seven types of data-flow propagation in TPG.

tion bitmap of edge $u \to v$ is compacted into a 64-bit unsigned integer, $\mathcal{M}_{u\to v}$, where the $(i*8+j)$th bit refers to the value of $map[i][j]$ and one multiplication can be optimized to $\Theta(n^2)$ with bitwise acceleration after an one-time mask initialization step, as shown in Algorithm 2. To improve efficiency, we also utilize memoization to store and reuse the results of previously performed multiplications.

TPG explicitly categorizes the semantics of different classes of data-flow propagation into seven types, each referring to one edge type in TPG, as illustrated in Table 1. Building upon this, each kind of instruction or data flow relationship is interpreted as a corresponding taint propagation type, with which edges in the TPG are constructed.

CFL-Reachability Analysis. Although taint analysis on the TPG is context-sensitive, the limited depth of recorded call contexts may still allow taint propagation paths that violate proper call-return matching. For example, a call from context $\langle c_1, c_2, c_3 \rangle$ to $\langle c_2, c_3, c_4 \rangle$ may return back to $\langle c_5, c_2, c_3 \rangle$, potentially generating false positives when implementations of different protocols invoke the same sequence of functions. To address this, we perform an additional CFL (Context-Free-Language) Reachability analysis on the TPG using a tabulation algorithm [36], leveraging the associativity of our matrix-like taint propagation operations. Specifically, we compute all taint summary edges $\mathsf{TaintSumm}(cs)$ from a call site node cs to return site nodes at the same call stack position, capturing both direct and indirect value flows while considering only feasible taint paths. During taint analysis, edges from function return points to return sites are ignored, and the computed summary edges are added into consideration. Moreover, a special case must be handled: before a taint reaches its first encountered call site, it should be able to propagate outside the parent function through function return points or indirect return points.

With the modeling of all possible data-flow transitions in the implementation of protocol stack by TPG, WAVED can perform taint analysis in a Meet-over-all-Paths (MOP) manner. Since a single forward program slice on the TPG can only capture the taint propagation of one 8-byte value, WAVED handles each input packet header byte separately. Specifically, it traverses the TPG for each packet field, incorporating computed CFL summaries to resolve inter-procedural CFL-Reachable taint propagation. At each TPG node n, WAVED aggregates the influence of all tainted bytes by mapping them back to their original offsets and recording the result in a bit-set associated with the node, denoted as $\mathsf{Taint}(n)$. In this way, the taint analysis *achieves MOP and soundness*. The byte-level, context-sensitive, and CFL-reachable taint propagation guaranteed by the TPG allows the analysis to more precisely assess the influence of the input packet, thereby effectively addressing Challenge II.

`Cmp`	$\mathbb{T}$(`Cmp`)
$a < b$ or $a \le b$	$\langle \mathsf{Taint}(a), \mathsf{Taint}(b), \phi \rangle$
$a > b$ or $a \ge b$	$\langle \mathsf{Taint}(b), \mathsf{Taint}(a), \phi \rangle$
$a = b$	$\langle \phi, \phi, \mathsf{Taint}(a) \vee \mathsf{Taint}(b) \rangle$
$a \neq b$	$\langle \phi, \phi, \phi \rangle$

Table 2: Mapping from `Cmp` to `Taint Tuple`-$\mathbb{T}()$. $\mathsf{Taint}(a)$ and $\mathsf{Taint}(b)$ refers to the tainted input bytes related to operator a and b, respectively.

4.3 Branch Impact Calculator

As highlighted in Challenge III, even with the byte granularity provided by our taint analysis, direction-insensitive taints fail to distinguish between strong and weak constraints on the same byte. This limitation makes it difficult to accurately measure weak verification along execution paths. Accordingly, we design Branch Impact Calculator (BIC), which leverages *direction-sensitive taint information*-`taint tuple` and `branch impact`-to measure branch and path constraints, and calculates `function summary` of each function-containing all possible intra-procedural paths with different constraints, to accelerate the final path discovery.

Definition 1 (`Taint Tuple` and `Branch Impact`)**.** *A* `taint tuple` $\langle L, G, E \rangle$ *is the basic unit of direction-sensitive taint information at the byte level, representing a single constraint,*

where each component L, G, E captures taints related to different comparison outcomes:

- *L: taints on bytes relevant to the* less-than *outcome (e.g., left operand of "$<$" or right operand of "$>$"),*
- *G: taints on bytes relevant to the* greater-than *outcome (e.g., right operand of "$<$" or left operand of "$>$"),*
- *E: taints on bytes relevant to* equality or bounded *outcomes (e.g., both operands of "$=$" or values constrained by range checks), which are considered as strong constraints.*

A `branch impact` *represents every possible constraint (i.e.,* `taint tuple`*) at a position. Given a set of possible* `taint tuple`*s at this position $\{\mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots, \mathcal{T}^{(n)}\}$, a* `branch impact` *$\mathcal{B}$ is defined as their disjunction:*

$$\mathcal{B} = \mathcal{T}^{(1)} \vee \mathcal{T}^{(2)} \vee \cdots \vee \mathcal{T}^{(n)}.$$

With Definition 1, each outcome of a `Cmp` Instruction can be mapped to a `taint tuple` by a mapping $\mathbb{T}()$, as shown in Table 2. BIC also considers the constraint aggregation in a direction-sensitive way: when the same taint byte appears in both of the unbounded one-sided constraints (L set and G set), the corresponding constraint is elevated to a range check, then the byte is subsequently included in the *bounded strong* constraint set E, so we have:

$$\langle L, G, E \rangle = \langle L \setminus (L \wedge G), G \setminus (L \wedge G), E \vee (L \wedge G) \rangle \quad (2)$$

Furthermore, to represent every possible constraint composition, we define the conjunction arithmetic for `taint tuple` and both the disjunction and conjunction arithmetics for `branch impact`:

$$\mathcal{T}_1 \wedge \mathcal{T}_2 = \langle \mathcal{T}_1.L \vee \mathcal{T}_2.L, \mathcal{T}_1.G \vee \mathcal{T}_2.G, \mathcal{T}_1.E \vee \mathcal{T}_2.E \rangle \quad (3)$$

$$\mathcal{B}_1 \vee \mathcal{B}_2 = \bigvee_{i=1}^{|\mathcal{B}_1|} \mathcal{T}_1^{(i)} \quad \vee \quad \bigvee_{j=1}^{|\mathcal{B}_2|} \mathcal{T}_2^{(j)} \quad (4)$$

$$\mathcal{B}_1 \wedge \mathcal{B}_2 = \bigvee_{\substack{i=1,\ldots,|\mathcal{B}_1| \\ j=1,\ldots,|\mathcal{B}_2|}} \left(\mathcal{T}_1^{(i)} \wedge \mathcal{T}_2^{(j)} \right) \quad (5)$$

To differentiate the constraint at each pair of if-else branch outcomes, we maintain a pair of `branch impact`-$\langle \mathcal{B}_1, \mathcal{B}_0 \rangle$ at each conditional branch instruction, respectively referring to the constraint of the `True` outcome and the `False` outcome, outperforming conventional direction-insensitive taint analysis which fails to make this distinction by considering both outcomes as tainted or untainted and leads to false positives. Then a path constraint can be easily measured: *the conjunction of* `branch impact`*s at all the control-dependent branch outcomes* after performing a control-dependency analysis.

Implicit control-flow tracking. In commodity kernel protocol stacks, critical path constraints are frequently encapsulated within helper functions or macros, where return values are dictated by internal checks, rendering the control flow syntactically opaque. List 2 illustrates the implementation of *tcp_sequence()* in Linux 6.8, which validates whether an incoming TCP sequence number lies within the receive window. Crucially, since all exit paths return untainted constant integers, standard taint analysis fails to track the implicit dependency between the input sequence and the validation result, inevitably resulting in false negatives. Conversely, merely employing implicit taint analysis lacks the direction-sensitive context required by our design, causing over-approximation and leading to false positives [9, 27].

```
static enum skb_drop_reason tcp_sequence(...)
{
    if (before(end_seq, tp->rcv_wup))
        return SKB_DROP_REASON_TCP_OLD_SEQUENCE;

    if (after(seq, tp->rcv_nxt + tcp_receive_window(tp)
        ))
        return SKB_DROP_REASON_TCP_INVALID_SEQUENCE;

    return SKB_NOT_DROPPED_YET;
}
```

Listing 2: *tcp_sequence()* implementation in Linux 6.8.

Therefore, we should design a method to *incorporate implicit taint into our direction-sensitive taint resolution*. An intuitive approach is to capture the path constraints associated with each return value. However, implementing this in static analysis is often impractical. First, the sheer number of possible return values can lead to path explosion. Second, these values frequently involve dynamic expressions that cannot be resolved statically. Nevertheless, we observe a critical pattern: the return values of such intermediate functions in the kernel protocol stack are primarily subjected to simple checks in the caller, such as equality to zero or null-pointer validation. For instance, *tcp_sequence()* uses 0 (*SKB_NOT_DROPPED_YET*) to indicate success, while *udp4_lib_lookup()* indicates a lookup failure simply by returning a NULL pointer. Therefore, we additionally associate each value with a pair of `branch impact`, $\langle \mathcal{B}_1, \mathcal{B}_0 \rangle$. $\mathcal{B}_1$ corresponds to the path constraints implied by a non-zero or non-null value, whereas $\mathcal{B}_0$ corresponds to the constraints implied by a zero or null value.

Calculation of `branch impacts`. Leveraging the TPG from Section 4.2, which models the complete data-flow of the protocol stack, we develop a technique to resolve `branch impacts` via backward resolution over the graph. BIC models seven types of instructions independently, including `Cmp`, `BinaryOP`, `UnaryOP`, `Phi`, `Select`, `Cast`, `Call`, as shown in Table 3. And for other types of instructions, BIC conservatively falls back to assign double-sided taint constraints on both $\mathcal{B}_1$ and $\mathcal{B}_0$.

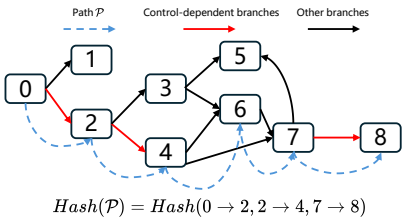


Figure 4: An example of path hashing for deduplication in `function summary`. Here, the hashing of the path $\mathcal{P}(0 \to 2 \to 4 \to 6 \to 7 \to 8)$ only depends on the control-dependent branch outcomes in the path ($4 \to 6$ and $6 \to 7$ are excluded since 7 is the immediate post-dominator of 4, and they have no control-flow impact on the rest of the path).

Among these, the `Phi` instruction presents the most significant challenge. Since it aggregates values based on the incoming block, it inherently embodies the transition from control-flow to data-flow, implying implicit constraints. Consequently, resolving the `branch impact` for a `Phi` node necessitates determining the each path constraint taken to reach the current block. To address this, *we dynamically construct a* `function summary` *during the resolution process*. Utilizing the intra-procedural transition path constraints to each predecessor of the `Phi` node $[\mathcal{B}_{path}(\mathcal{P}_1), \ldots, \mathcal{B}_{path}(\mathcal{P}_n)]$, we can precisely derive the path constraints from the function entry to each predecessor of the `Phi` node $[\mathcal{B}_{path}(\mathcal{P}_1), \ldots, \mathcal{B}_{path}(\mathcal{P}_n)]$, the implicit constraints behind the `Phi` value can be obtained.

Definition 2 (`Function Summary`)**.** *A* `Function Summary` $\mathcal{S}(ctx, fun, bb)$ *represents the set of pairs of intra-procedural basic block transition paths and their path constraints* $[(\mathcal{P}_1, \mathcal{B}_{path}(\mathcal{P}_1)), (\mathcal{P}_2, \mathcal{B}_{path}(\mathcal{P}_2)), \ldots, (\mathcal{P}_n, \mathcal{B}_{path}(\mathcal{P}_n))]$ *ending at* basic block *bb in* function *fun under* calling context *ctx. Each path* $\mathcal{P}_i$ *contains a sequence of transitions* $(bb_i^{(j)}, edge_i^{(j)})$*, where* $bb_i^{(j)}$ *is the j-th control-dependent basic block in* $\mathcal{P}_i$ *and* $edge_i^{(j)}$ *indicates the taken outcome.*

To compute the `branch impact` and `function summary` simultaneously, BIC performs a breadth-first traversal from the function entry. Loop edges are ignored since they always incur stronger constraints, thus we can process basic blocks in topological order. Therefore, the `branch impact` of the current termination instruction is resolved based on the current results, after which new intra-procedural paths to successor basic blocks—integrated with merged path constraints—are calculated and updated into the `function summary`. This design allows BIC to reuse previously computed paths and constraints, ensuring that each node is considered at most once, therefore achieving efficiency.

Although `function summary` calculation is restricted to an intra-procedural scope, path explosion remains a significant challenge in complex functions. To address this, we implement a path deduplication strategy leveraging control dependency analysis. During traversal, BIC maintains a stack to track active `branch impacts`. Crucially, a branch is popped from the stack upon reaching its immediate post-dominator. This mechanism guarantees that the stack contains only the constraints that strictly control the current execution path. By hashing the stack's content, we can identify and deduplicate equivalent path states reaching the same basic block, as illustrated in Figure 4. The overall process of the `function summary` calculation is shown in Algorithm 3.

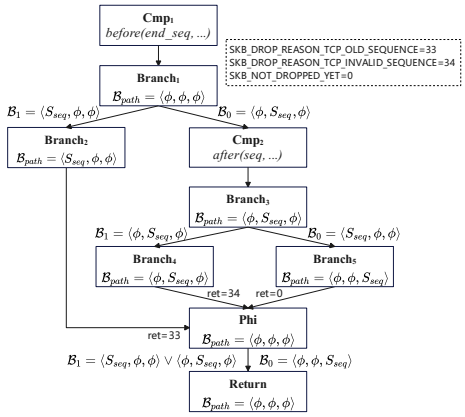


Figure 5: Example of `branch impact` calculation in *tcp_sequence()* in Linux 6.8. S_{seq} refers to the set of byte indices of TCP sequence number in the input packet.

Figure 5 illustrates how BIC calculates the `branch impact` of the return value of *tcp_sequence()* (Linux 6.8). Through our implicit control-flow tracking mechanism, BIC accurately extracts the control-flow constraints underlying the return values, deriving $\mathcal{B}_0$ for accepting a packet and $\mathcal{B}_1$ for dropping it. Unlike previous work (e.g., [9]), WAVED can aggregate constraints of one-sided comparisons (*after()* and *before()* comparisons), enabling it to distinguish between strong in-window checks and weak out-of-window checks.

4.4 Weak Path Identifier

Finally, the Weak Path Identifier (WPI) leverages the intra-procedural paths in `function summaries` computed by BIC, conjoining them together with constraints along each inter-procedural execution path to risky operations, to determine whether a exploitable path is weakly constrained. It then reports the weak-verification paths in a detailed, structured format—including each control-dependent branch and the `branch impact` of the taken outcome—to the user.

To effectively compose the pre-computed `function summaries` into execution paths, WPI first constructs a *Summary Graph* based on the `function summaries`. In this graph, each node represents a context-sensitive call graph node (a function in one context), and each edge corresponds

Instruction	Condition	Resolution Rule for $\langle \mathcal{B}_1, \mathcal{B}_0 \rangle$	Hints
`Cmp`$(op_1 \diamond op_2)$	Const Boolean Operator Other kinds	$\langle \mathcal{B}_1(\text{Eval}(op_1 \diamond op_2)), \mathcal{B}_0(\text{Eval}(op_1 \diamond op_2)) \rangle$ $\langle \mathbb{T}(op_1 \diamond op_2),\ \mathbb{T}(op_1 \diamond' op_2) \rangle$	– $\diamond'$ is the complement of $\diamond$, (e.g., $<$ to $>$, $=$ to $\neq$)
`Binary`$(op_1 \diamond op_2)$	Boolean Operation Non-Boolean	$\langle \mathcal{B}_1(op_1), \mathcal{B}_0(op_1) \rangle \diamond \langle \mathcal{B}_1(op_2), \mathcal{B}_0(op_2) \rangle$ $\langle \mathcal{T}_{res}, \mathcal{T}_{res} \rangle$	– $\mathcal{T}_{res} = \langle \phi, \phi, \mathsf{Taint}(res) \rangle$
`Unary`$(\neg op)$	–	$\langle \mathcal{B}_0(op),\ \mathcal{B}_1(op) \rangle$	`FNeg`
`Phi`$(\langle bb_1, op_1 \rangle, \ldots, \langle bb_n, op_n \rangle)$	–	$\mathcal{B}_1 = \bigvee_{i=1}^{n} (\mathcal{B}_{path}(bb_i) \wedge \mathcal{B}_{jump}(bb_i) \wedge \mathcal{B}_1(op_i))$ $\mathcal{B}_0 = \bigvee_{i=1}^{n} (\mathcal{B}_{path}(bb_i) \wedge \mathcal{B}_{jump}(bb_i) \wedge \mathcal{B}_0(op_i))$	$\mathcal{B}_{path}(bb_i)$: the path constraints to bb_i $\mathcal{B}_{jump}(bb_i)$: the jump condition from predecessor bb_i
`Select`$(cond, op_T, op_F)$	–	$\mathcal{B}_1 = (\mathcal{B}_1(cond) \wedge \mathcal{B}_1(op_T)) \vee (\mathcal{B}_0(cond) \wedge \mathcal{B}_1(op_F))$ $\mathcal{B}_0 = (\mathcal{B}_1(cond) \wedge \mathcal{B}_0(op_T)) \vee (\mathcal{B}_0(cond) \wedge \mathcal{B}_0(op_F))$	–
`Cast`(op)	–	$\langle \mathcal{B}_1(op),\ \mathcal{B}_0(op) \rangle$	–
`Call`$(func(arg_0, \ldots))$	–	$\langle \mathcal{B}_1(ret_{func}),\ \mathcal{B}_0(ret_{func}) \rangle$	Inter-procedural Copy

Table 3: Resolution rules for `branch impact` $\langle \mathcal{B}_1, \mathcal{B}_0 \rangle$ on TPG.

to a basic block path within the `function summary`. The whole procedure is represented in Algorithm 4. Subsequently, a depth-first traversal is performed on the *Summary Graph*, initiating from the entry node (i.e., the packet receiving entry). During this process, the `branch impact` of each traversed edge is accumulated to derive the final path constraint. WPI further reduces the path space and filters out weakly constrained ones with three optimization methods—*header activation*, *constraint deduplication* and *strength filtering*—that leverages precomputed `branch impacts`.

Header Activation. Although we employ CFL-Reachability analysis, false positives persist due to limited call stack depth maintained in the analysis, particularly when taint information propagates across protocols via deep-stacked shared interfaces. To mitigate this, we introduce a path-sensitive taint activation mechanism: tracking for protocol fields is enabled only when execution enters the corresponding protocol context. For example, entering the TCP input function (e.g., `tcp_v4_rcv()` in Linux) triggers tracking for TCP header fields, preventing unnecessary over-tainting. Accordingly, during the *Summary Graph* traversal, the `branch impact` of each edge is re-evaluated, and constraints associated with inactive contexts are excluded.

Constraint Deduplication. We further observed that many intra-procedural paths yield equivalent constraints, often stemming from fast-path/slow-path implementations. Reporting all such paths leads to a path explosion that overwhelms manual analysis. To address this, we refine the path hashing mechanism during the *Summary Graph* traversal to deduplicate these paths. Specifically, we re-calculate the hash for each sub-path derived from the `function summary`. If a control-dependent branch outcome has no contribution to the current accumulated `branch impact`—requiring no new knowledge of the attacker even in the case of a weak one-sided comparison—it is deemed uninteresting and excluded from the path hash calculation. Nevertheless, since it remains control-dependent of the path destination, such a branch will still be included in the result to help user better understand the path. Consequently, paths with distinct constraints are retained, while paths yielding identical constraints—sharing a common path constraint prefix—are deduplicated.

Strength Filtering. Upon reaching a labeled risky operation, WPI identifies the weakest constraint on the path—defined as the constraint whose associated `taint tuple` yields the minimum number of strongly constrained bytes (i.e., the minimum $|E|$), and flags the path if this minimum size falls below a predefined threshold. Besides counting number of strongly constrained bytes, WPI also exclude paths with strong constraints of highly secure fields (e.g., TCP/DCCP seq/ack, SCTP vtag), which are usually impossible for an off-path attacker to eavesdrop or brute-force. After filtering, the path is considered weakly constrained and logged for user inspection.

5 Implementation

In this section, we briefly present implementation details of the main components in WAVED.

Protocol-Oriented Pointer Analysis. The Protocol-Oriented Pointer Analysis is implemented as an independent module within SVF's WPA library, built on version 3.0 (commit `a68b293` [38]). To support byte-offset-based `GEP` resolution, we modify and extend SVF's low-level interfaces responsible for processing `GEP` instructions, and filter out variant `GEP`s that make SVF treat base objects as field-insensitive, inducing excessive false positives in large-scale programs. Additionally, we extend the ExtAPI module to properly handle external memory functions in kernel code. To further reduce false positives arising from overlapping field offsets in C `union` structures, we employ a Clang plugin to transform `union` definitions into `struct`s. For robustness, transformations are skipped for network packet headers and system header files. The final implementation consists of an SVF patch integrated with the framework and a Clang plugin, totaling approximately 4.9k lines of C++ code.

Byte-Granularity Taint Analysis. Unlike SVFG in SVF which only considers pointer-related instructions, all program instructions are included during TPG construction. To improve efficiency, CFL-Reachability analysis is restricted to

TPG nodes on an object-granularity simple forward slice rooted at taint sources. This component is implemented with roughly 4.1k lines of C++ code based on SVF and LLVM.

Branch Impact Calculator & Weak Path Identifier. The Branch Impact Calculator and Weak Path Identifier are implemented as external SVF modules. The final output enumerates all potential weakly-validated paths along with the detailed branch constraints, presented in an HTML report for easy user inspection. The combined implementation comprises approximately 1.6k lines of C++ code.

In total, the complete implementation of WAVED consists of around 12.4k lines of C++ code.

6 Evaluation

In this section, we evaluate the efficiency, efficacy, precision and reduction of WAVED by applying it to the IPv4/IPv6 implementation of three widely-used open source kernel versions, including Linux 5.15, Linux 6.8 and FreeBSD 14.1. For Linux kernel, we test ICMP, TCP, UDP, SCTP and DCCP protocol in both IPv4 and IPv6. Since FreeBSD does not support DCCP, for FreeBSD kernel, we test ICMP, TCP, UDP and SCTP protocol in both IPv4 and IPv6.

Kernel	Code Coverage	Total TPG Node
Linux5.15-IPv4	107,536 LoC	17,544,618
Linux5.15-IPv6	135,033 LoC	20,318,792
Linux6.8-IPv4	114,602 LoC	14,171,801
Linux6.8-IPv6	141,514 LoC	16,270,503
FreeBSD14.1-IPv4	94,149 LoC	18,881,472
FreeBSD14.1-IPv6	96,103 LoC	32,816,913

Table 4: Analysis Coverage of WAVED.

6.1 Evaluation Setup

WAVED is deployed on a server with Intel Xeon Gold 5418Y processor and 500GB RAM, and *runs automatically without human intervention* after configuration. To prevent control-flow interleaving between IPv4 and IPv6, we compile them separately for each kernel. Specifically, the IPv4 and IPv6 implementations are isolated via distinct source files or conditional compilation flags. We refer to the resulting variants as *IPv4-kernel* and *IPv6-kernel* respectively. Hence, six independent kernels are obtained. To maximize the code coverage, we use `allyesconfig` for Linux and `GENERIC` for FreeBSD. The analysis coverage of each kernel is shown in Table 4.

Taint source marking. We mark each byte of incoming packet headers (ICMP, ICMPv6, TCP, UDP, SCTP, DCCP) as taint sources, excluding IPv4/v6 headers and fields unlikely to influence checks (e.g., `len`, `doff`, `csum`). For ICMPv4/v6 error packets, inner headers are modeled separately from outer ones.

Risky operation labeling. Based on prior studies [9, 16, 17, 20], two main exploiting patterns are considered in WAVED: FNHE/routing cache poisoning and packet injection. For cache poisoning, we label `update_or_create_fnhe()` for Linux IPv4, `rt6_insert_exception()` for Linux IPv6, and `rib_add_redirect()` plus `tcp_hc_updatemtu()` for FreeBSD. For packet injection, protocol data-receiving points are included. Finally, we also investigate the function `tcp_send_challenge_ack()`, though it is not a direct security violation operation which leads to risks, but its triggering condition involves intricate in-window and out-of-window checks. This case study further demonstrates the effectiveness of direction-sensitive taints leveraged by WAVED.

Analysis initialization. In order to support flow-sensitive analysis, we add a `main()` function to each tested kernel, where network initialization functions and the packet receive function (`ip_rcv()`/`ipv6_rcv()` for Linux, `ip_input()`/`ip6_input()` for FreeBSD) are sequentially called. The reporting threshold is set to 8 bytes, calibrated to the most rigorous known weak verification case (4 bytes for src/dst ports and 4 bytes for the target gateway address) [20].

6.2 Analysis Results

Running time. On our testing platform, WAVED takes 22 minutes and 30 minutes to analyze the IPv4 and IPv6 implementations of Linux 5.15, 11 minutes and 21 minutes for Linux 6.8, and 42 minutes and 125 minutes for FreeBSD 14.1, *demonstrating the significant efficiency of our MOP analysis on such large-scale programs*.

Uncovered vulnerabilities. WAVED identified *a total of 19 weak-verification vulnerabilities* in the IPv4 and IPv6 implementation of Linux and FreeBSD (16 in Linux and 3 in FreeBSD). In Linux 5.15 and 6.8, 12 of 16 are newly discovered, 10 for routing cache poisoning and 2 for packet injection. In FreeBSD 14.1, 2 of 3 are newly discovered, both poisoning the routing cache. We manually validated all of the *14 newly found vulnerabilities* in a controlled local environment, confirming that each can be practically exploited. The issues reported by WAVED are listed in Table 5 and Table 6.

6.3 Findings

In this section, the summarization of the weak-verification paths identified by WAVED is given.

6.3.1 Findings in Linux

Cache Poisoning. In both *Linux5.15-IPv4* and *Linux6.8-IPv4*, WAVED identifies 34 MTU-poisoning paths and 12 gateway-poisoning paths. Each set of paths can be consolidated into 6 unique reports. The multiplicity of reports per unique report arises from runtime-unreachable paths, primarily due to repetitive type/state checks. For example, *Type* and *Code* fields of an

#	Protocol	Packet Form	Attack Constraints	Attack Effect	Affected System
1	ICMPv4	Redirect [Echo Reply]	No extra constraint	gw cache poisoning	Linux 5.15 & Linux 6.8
2		Redirect [UDP]	UDP sport match (UDP dport match when connected)		
3		Redirect [IP]	RAW socket of inner protocol		
4		Redirect [Echo Request]	Ping ident match		
5		Fragment Needed [Echo Reply]	No extra constraint	pmtu cache poisoning	
6		Fragment Needed [UDP]	UDP sport match (UDP dport match when connected)		
7		Fragment Needed [IP]	RAW socket of inner protocol		
8		Fragment Needed [Echo Request]	Ping ident match		
9		Redirect [IP]	No extra constraint	gw cache poisoning	FreeBSD 14.1
10	ICMPv6	NDISC Redirect [ICMPv6]	No extra constraint		Linux 5.15 & Linux 6.8
11		NDISC Redirect [UDPv6]	UDP sport match (UDP dport match when connected)		
12		NDISC Redirect [IPv6]	RAWv6 socket of inner protocol RECVERR option set or connected		
13		Packet Too Big [ICMPv6]	No extra constraint	pmtu cache poisoning	
14		Packet Too Big [UDPv6]	UDP sport match (UDP dport match when connected)		
15		Packet Too Big [IPv6]	RAWv6 socket of inner protocol RECVERR option set or connected		
16		NDISC Redirect [IPv6]	No extra constraint	gw cache poisoning	FreeBSD 14.1
17		Packet Too Big [IPv6]	No extra constraint	pmtu cache poisoning	

Table 5: Cache poisoning vulnerabilities uncovered by WAVED. The packet form `outer pkt [inner pkt]` means `inner pkt` is encapsulated in `outer pkt`. All ICMP/ICMPv6 error messages assume outer pkt dst = victim, inner pkt src = victim; redirect attacks require outer pkt src = victim's gateway and only Linux ICMPv4 checks if the target gateway exists on the local link before accepting; PMTU attacks require a smaller PMTU value than the original one. **No color**: new. **Yellow**: previously reported.

#	Protocol	Packet Type	Attack Constraints	Attack Effect	Affected System
18	DCCPv4	RST /CLOSEREQ /CLOSE	sock in DCCP_REQUESTING DCCP port match	Denial of Service	Linux 5.15 & Linux 6.8
19	DCCPv6				

Table 6: Packet injection vulnerabilities uncovered by WAVED (both are new).

ICMP header may be checked multiple times along a single execution path. Due to the limitations of static analysis, WAVED cannot determine the precise runtime values at every program point, resulting in duplicate paths. Among the 12 unique reports, 8 are True Positives (TPs) and 4 are False Positives (FPs). The FPs are caused by redundant branches: A TCP or DCCP error handler does not require an in-window sequence number in `TCP_LISTEN` or `DCCP_REQUESTING`/`DCCP_LISTEN` states. However, in these cases, the socket is obtained from `__inet_lookup_established()`, only connection-established sockets are considered, misleading WAVED to count in them. We suggest Linux developers to remove these redundancies, which may introduce inconsistencies.

In both *Linux5.15-IPv6* and *Linux6.8-IPv6*, WAVED identifies 5 MTU-poisoning paths and 6 gateway-poisoning paths. Each set of paths can be consolidated into 5 unique reports. Among these 10 unique reports, 6 are TPs and 4 are FPs. The FPs are also brought by the same redundant branches discussed in the previous paragraph.

Packet Injection. For UDPv4/v6, WAVED identifies 4 paths in both Linux 5.15 and 6.8, covering unicast and multicast inputs; all of them are TPs, as only port checks are required to locate the corresponding socket. For TCPv4/v6, no paths bypass the double-sided sequence number verification. Similarly, for

SCTPv4/v6, the mandatory `sctp_vtag_verify()` function ensures that no weakly constrained path exists.

For DCCPv4/v6, WAVED reports 7 potential weak paths each, all passing through the same branch `if (sk->sk_state != DCCP_REQUESTING && dccp_check_seqno(sk, skb))`, assuming `sk_state == DCCP_REQUESTING` to bypass the strongly constrained in-window sequence number check. After consolidating the paths, both DCCPv4 and DCCPv6 result in 4 unique reports, comprising 3 TPs and 1 FP. The only FP arises from repetitive state checks (WAVED enters a later branch assuming the socket state is `DCCP_RESPOND`, which is inconsistent with the previous checked state, due to the limitation of static analysis), while the 3 TPs correspond to a scenario in `DCCP_REQUESTING` state where Linux does not verify the incoming sequence number for `DCCP_PKT_RESET`, `DCCP_PKT_CLOSEREQ`, and `DCCP_PKT_CLOSE`, creating a narrow attack window in both IPv4 and IPv6 implementations of Linux 5.15 and 6.8.

6.3.2 Findings in FreeBSD

Cache Poisoning. In *FreeBSD14.1-IPv4*, WAVED identified 1 weak-verification path to `rib_add_redirect()`, revealing that the kernel does not validate the encapsulated packet in ICMP `Redirect`s. Since ICMP-encapsulated `Fragment Needed`s are ignored and UDP-encapsulated ones only notify the upper layer, while other protocols enforce strong checks, no paths to `tcp_hc_updatemtu()` were found.

In *FreeBSD14.1-IPv6*, WAVED found 1 weak path to `rib_add_redirect()` due to the absence of inner header validation, and 1 risky path to `tcp_hc_updatemtu()`.

Packet Injection. WAVED detected 2 and 4 injection paths for UDPv4 and UDPv6, respectively; all are TPs, covering both unicast and multicast inputs, and relying solely on port checks. SCTPv4/v6 exhibited no weak paths due to mandatory 32-bit vtag verification. For TCPv4/v6, 4 paths were reported for each, which are consolidated into a single false positive because WAVED cannot track the runtime socket state, as discussed in Section 6.3.1.

6.4 Precision and Reduction

Table 7 shows the weakly constrained paths reported by WAVED. Profiting from precise constraint modeling and path deduplication, WAVED successfully detects a total of 43 unique weakly constrained paths in the kernel protocol stack while generating only a few FPs. Most FPs (8 of 12) arise from redundant branches incorrectly implemented in Linux (discussed in Section 6.3.1), misleading WAVED to explore infeasible branches. These FPs share a common feature—all pass through the same redundant branch. As WAVED outputs all of the control-dependent branches along each identified path, users can easily identify and skip them by noticing the identical structure, and another feasible way is simply removing it from the source code and reanalyzing (thanks to the high efficiency of WAVED). For the remaining 4 FPs, we verified that they are all caused by repetitive type/state check, which is due to a common limitation in static analysis. Overall, WAVED achieves considerably high precision in identifying weak-verification paths that expose vulnerabilities.

Kernel	Type	Total Paths	Unique Reports	TP	FP
Linux 5.15 -IPv4 & Linux 6.8 -IPv4	C-MTU	34	6	4	2
	C-GW	12	6	4	2
	P-UDP	4	2	2	0
	P-DCCP	7	4	3	1
Linux 5.15 -IPv6 & Linux 6.8 -IPv6	C-MTU	5	5	3	2
	C-GW	6	5	3	2
	P-UDP	4	2	2	0
	P-DCCP	7	4	3	1
FreeBSD 14.1 -IPv4	C-GW	1	1	1	0
	P-UDP	2	2	2	0
	P-TCP	4	1	0	1
FreeBSD 14.1 -IPv6	C-MTU	1	1	1	0
	C-GW	1	1	1	0
	P-UDP	4	2	2	0
	P-TCP	4	1	0	1

Table 7: Weak-verification paths identified by WAVED. Only path types detected at least once are included. **C**: cache poisoning. **P**: packet injection. **GW**: gateway.

To evaluate the reduction of WAVED on path discovery, we performed an ablation study in Table 8, by enabling and disabling the optimizations introduced in Section 4.4. Specifically, we compared the results of the full system against configurations where the *strength filter* mechanism was disabled, and where the direction-sensitive taint calculation was replaced by a traditional implicit taint analysis as in [9, 27]. The results demonstrate that WAVED significantly reduces the path space via *constraint deduplication* (95% reduction for *Linux 5.15-IPv4/IPv6* and *Linux 6.8-IPv4*, 96% for *Linux 6.8-IPv6*, 71% for *FreeBSD 14.1-IPv4* and 61% for *FreeBSD 14.1-IPv6* in average). Although the protocol stack implementations have undergone substantial changes from *Linux 5.15* to *Linux 6.8* (resulting in different path numbers in implicit taint analysis), the fundamental types of path constraints remain invariant. The *constraint deduplication* mechanism successfully captures this invariance, yielding the same number of paths after deduplication. Furthermore, *strength filtering* eliminates strongly constrained paths while retaining those with weak verifications, resulting in average reductions of 52% for *Linux-IPv4*, 55% for *Linux-IPv6*, 58% for *FreeBSD-IPv4*, and 48% for *FreeBSD-IPv6*, thereby effectively improving the efficiency of the final inspection.

6.5 Analysis besides Direct Security Violation

As discussed in Section 6.1, besides direct security violation, we also evaluate WAVED on `tcp_send_challenge_ack()` in *Linux6.8-IPv4*, the trigger of a well-known side-channel

Kernel	Type	WAVED	w/o SF	w/o DT
Linux 5.15	C-MTU	34 / 5	42 / 8	13,416 / 144
	C-GW	12 / 6	18 / 9	4,680 / 192
	P-TCP	0 / 0	7 / 12	31,376 / 58,720
	P-UDP	4 / 4	4 / 4	16 / 42
	P-SCTP	0 / 0	8 / 6	272 / 76
	P-DCCP	7 / 7	18 / 18	720 / 720
Linux 6.8	C-MTU	34 / 5	42 / 8	7,332 / 168
	C-GW	12 / 6	18 / 9	2,652 / 216
	P-TCP	0 / 0	7 / 12	* / *
	P-UDP	4 / 4	4 / 4	16 / 42
	P-SCTP	0 / 0	8 / 6	200 / 108
	P-DCCP	7 / 7	18 / 18	720 / 3,600
FreeBSD 14.1	C-MTU	0 / 1	1 / 2	3 / 4
	C-GW	1 / 1	1 / 1	1 / 1
	P-TCP	4 / 4	46 / 46	* / *
	P-UDP	2 / 4	2 / 4	20 / 9
	P-SCTP	0 / 0	2 / 2	240 / 160

Table 8: Reduction of WAVED in path discovery. To save space, the results are presented in the format of IPv4 / IPv6. **w/o SF**: without *strength filtering*. **w/o DT**: without direction-sensitive taint calculation. ***** : over 100,000.

attack [6, 7], to further demonstrate the effectiveness of direction-sensitive taint analysis. WAVED outputs 8 paths (all TPs), categorized into 3 classes: i) The PAWs check fails when receiving a `SYN` packet, triggering a `SYN` challenge. The reported constraint is port matching; ii) The sequence number is out-of-window upon receiving a `SYN` packet. WAVED outputs two possible constraints—left- or right-sided sequence number taint, corresponding to an out-of-window sequence number—together with the port matching; iii) A `RST` packet with a non-matching sequence number triggers a challenge `ACK`. Only the port matching is reported as a constraint, since non-equal comparison of sequence number is considered weak and ignored by WAVED.

There is another way triggering a challenge `ACK`-an incoming `ACK` with an out-of-window `ACK` number. However, reaching this check requires a strongly critical constraint-an in-window sequence number. WAVED correctly omits this case, focusing only on weakly constrained paths.

This emphasizes that WAVED leverages direction-sensitive taint analysis to precisely model the distinct constraint strengths of different branch outcomes, consequently reducing false positives and optimizing path exploration.

7 Case Study

In this section, we provide details on the weak-verification vulnerabilities WAVED uncovered (Table 5 and Table 6 show the details of these vulnerabilities, which will be referenced by their IDs hereafter). Specifically, we categorize them by their attack constraints. We also discuss about their potential hazards and our suggestions to mitigate them.

7.1 Cache poisoning

Zero-Checked Vulnerabilities. WAVED identifies seven cache poisoning cases imposing no extra constraints beyond the basic requirements of forging ICMP/ICMPv6 error packets (#1, #5, #10, #13 in Linux and #9, #16, #17 in FreeBSD), termed *zero-checked*, constituting the weakest cases.

For those reported in Linux, the attack simply requires an ICMP/ICMPv6 error packet encapsulated with an ICMP/ICMPv6 packet. According to RFC 792 [34], ICMP errors are intended solely to report datagram errors, and RFC 4443 [12] specifies that an ICMPv6 error must not trigger another ICMPv6 error. Therefore, the safest mitigation is to ignore them.

For all the three reported vulnerabilities in FreeBSD, we found that they are all zero-checked, allowing an attacker to freely craft the inner IP/IPv6 packet-any valid structure is sufficient-encapsulated within the outer ICMP/ICMPv6 error packet. Since exploitation of these vulnerabilities can manipulate the global routing cache in the kernel, it is crucial to perform prior validation before updating the cache.

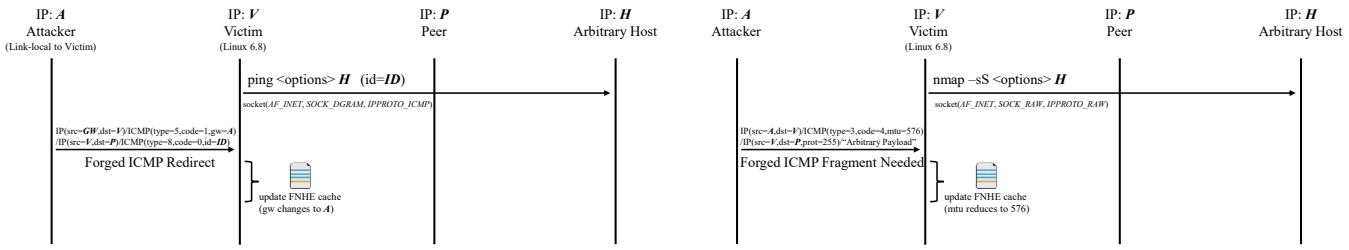
```
if (icmp6type == ICMP6_PACKET_TOO_BIG) {
    ......
    icmp6_mtudisc_update(&ip6cp, 1);
}
......
    ip6_ctlprotox[nxt](&ip6cp);
```

Listing 3: Code snippet in FreeBSD 14.1. The MTU update logic is before calling control functions of upper layers.

Interestingly, in the IPv4 implementation of FreeBSD 14.1, receiving an ICMP `Fragment Needed` packet first triggers the control input functions of the upper-layer protocols for validation before updating the MTU, which is safe. In contrast, the IPv6 implementation is opposite in logic (Listing 3), suggesting a potential inadvertent oversight.

UDP-based Vulnerabilities. Another four cases occur in the validation of ICMP/ICMPv6 error packets encapsulating a UDP packet (#2, #6, #11, #14), where the check depends only on the port numbers. Particularly, if the corresponding UDP socket is unconnected, the destination port is treated as a wildcard; otherwise, both source and destination ports must match. The IPv4 variants of these cases have been reported in prior studies [17, 20], where attackers could exploit them to launch Denial-of-Service (DoS) attacks or hijack active connections. Since the server's port is typically known, an attacker only needs to enumerate at most 2^{16} possible packets even when the socket is connected. A practical mitigation, following FreeBSD's handling of ICMP `Fragment Needed` packets encapsulating UDP, is to notify the application layer of such events while avoiding modifications to kernel.

Raw-Socket-based Vulnerabilities. Four cases are based on an existing raw socket and occur only in Linux (#3, #7, #12, #15). When an ICMP/ICMPv6 error packet triggered by an IP/IPv6 packet arrives, the kernel forwards it to the corre-



(a) A ping-based redirect attack example (#4).

(b) A raw-based IP fragmentation attack example (#7).

Figure 6: Sample PoCs for two newly found cases #4 and #7.

sponding upper-layer protocol for validation while simultaneously delivering it to the raw error handlers associated with the protocol number (255 for raw sockets with `IPPROTO_RAW`). For compatibility reasons, raw sockets do not maintain upper-layer protocol secrets, allowing access based solely on the IP/IPv6 address to locate the socket and modify the associated routing cache. Many professional network performance and audit tools (e.g., NMap [14], ZMap [40]) typically instantiate raw sockets in the background to perform highly customized active probing, Figure 6b shows a sample PoC sample for #7 when a victim is utilizing NMap to perform a TCP SYN scan to an arbitrary host, triggering fragmentation of peer-bound traffic from victim via a forged `Fragment Needed` packet.

In Linux IPv6, a stricter check is applied: only raw sockets with the `RECVERR` option enabled or connected to a peer address accept error messages. However, the `RECVERR` option is intended to notify applications of errors, and directly modifying the routing cache based on these messages is unsafe. Even if an ICMP raw socket is connected, the lack of additional verification allows an attacker to initiate attacks as soon as the peer's address is known. Since raw sockets are commonly used for debugging and testing, a practical mitigation is to prevent error handlers from modifying the routing cache.

Ping-based Vulnerabilities. Two cases are triggered by an ICMP error packet encapsulating an ICMP `Echo Request` (ping) and exist only in the IPv4 implementation of Linux (#4, #8). In this scenario, an attacker must correctly guess the ping identifier (ident) before the corresponding ping socket is closed. Although the attack window is short—typically the lifetime of a ping process—this guess is feasible. We further observe that the identifier information can be directly read from `/proc/net/icmp` without superuser authority in Linux, meaning that a low-privileged puppet controlled by the attacker can obtain it easily [10, 18, 35, 42]. Even without such puppet, the enumeration space is only 2^{16}, making the attack still practicable. A sample PoC for #4 is shown in 6a, where the attacker diverts traffic intended for the peer toward himself to achieve a full logical Man-in-the-Middle hijack.

The primary issue is that the kernel does not store the destination addresses of sent ping packets, leaving the ident as the sole secret for verification. However, maintaining the destination address for every ping is costly and impractical. Since ping packets are usually small and unlikely to be fragmented, any resulting `Fragment Needed` messages should be safely ignored. Similarly, `Redirect` messages, which rarely occur in practice, should also be disregarded without risk.

7.2 Packet Injection

WAVED also discovers two vulnerabilities of packet injection type (#18, #19). They are found in the DCCP implementation of Linux due to implementation flaw. When a socket is in `DCCP_REQUESTING`, no further check is applied to an incoming `RESET/CLOSEREQ/CLOSE`, violating RFC 4340 [25], making attackers able to terminate a connection request via a forged `Reset` filling with arbitrary 48-bit sequence number before the server replies a `Response`. We confirmed that this issue has persisted since Linux 3.1. However, previous work on packet injection [9] failed to detect it, likely due to the blind spots introduced by their human-assisted layered analysis. In contrast, WAVED employs a soundy, fully automated analysis that avoids such limitations. We argue that, for control packets which may lead to the termination of a connection, a strong sequence number check must be applied.

8 Discussion

Despite its capabilities, WAVED has several limitations.

- WAVED relies on static taint analysis and cannot capture the actual runtime state at each program point, which may produce FPs when different values of the same state are checked sequentially. Symbolic execution could mitigate this issue but it is comparatively time-consuming: a prior work [29] analyzing only TCP and UDP required over 1000 CPU hours. Nevertheless, by integrating loop mitigation and packet header activation, WAVED achieves soundness while limiting reported risky paths and reducing manual inspection effort.
- Since semantic vulnerabilities are inherently hard to define, requiring detailed attack patterns, WAVED only tests two known semantically risky patterns currently. However,

the configurable interface of WAVED allows users to add new patterns as they are discovered. Thus, comprehensive analysis of the new vulnerability class can be automated.

- The current version of WAVED does not support tunneling protocols (e.g., GRE, IPSec, VPN) due to their complex header encapsulation, and we leave this for future work.

Mitigations. Exploitation of weak verifications typically relies on IP spoofing, which is feasible in local networks or unfiltered ASes, as shown by prior TCP/IP attacks [6, 16–18, 20, 21, 28, 30]. The most effective defense is for routers to drop packets with invalid source addresses. Even if filtering is unavailable, according to RFC 792 and RFC 4443, protocol stacks should carefully validate the ICMP/ICMPv6 error messages according the upper layer secrets. For stateless protocols like ICMP, ICMPv6, and UDP, which lack socket-specific secrets in kernel caches, the safest approach is notifying upper-layer applications without modifying shared kernel state. In addition, strong checks must be applied to the incoming control packets of each protocol. For gateway updates, we suggest verifying the new gateway's reachability to mitigate the risk of traffic blackholing and subsequent DoS attacks.

Scalability. With the generic pointer and taint analysis framework, WAVED can be extended to any other protocol stacks, while flexible integrating additional protocols and specifying various semantically risky operations.

Responsible disclosure. We have reported our findings to the Linux and FreeBSD security teams, including detailed proof-of-concept exploits for all identified vulnerabilities, and we are working with them to develop and deploy patches.

9 Related Work

Off-path TCP/IP protocol suite attacks. The TCP/IP protocol suite has long been a target of off-path attacks [2, 3]. Feng *et al.* uncovered vulnerabilities in mixed IPID assignments in modern Linux systems, enabling TCP off-path hijacking, and further showed that semantic gaps in UDP could be exploited via crafted ICMP `Redirects` [15–17]. Gilad *et al.* exploited global IPID counters in older OSes to infer active TCP connections and perform off-path injection attacks poisoning HTTP and Tor traffic [22–24]. Yang *et al.* revealed that NAT implementations may vulnerable to forged TCP `RSTs`, allowing TCP hijacking by manipulating sequence numbers [41].

Other off-path vectors include exploiting side channels in the TCP challenge `ACK` mechanism [6, 7], leveraging middleboxes or malicious applications to infer TCP sequence numbers [35], and timing-based Wi-Fi attacks that inject TCP data via cached malicious content [10]. For UDP, forged ICMP error messages have been used to infer ephemeral UDP source ports, enabling DNS cache poisoning [28, 30]. Collectively, these studies illustrate the broad spectrum of off-path attacks exploiting shared states, weak verification, or side channels across TCP/IP protocols.

Principled semantic flaw detection in protocol stacks. Several principled approaches target the discovery of semantic vulnerabilities in protocol suites, focusing primarily on side channels. Cao *et al.* used model checking to detect non-interference violations in TCP connections [8]. Yu *et al.* combined taint analysis, entropy theory, and automated patching to iteratively uncover all side channels potentially observable by attackers [42]. Man *et al.* introduced SCAD, leveraging selective symbolic execution to identify non-interference violations across multiple targets while reducing FPs [29]. Beyond side channels, Chen *et al.* proposed PacketGuardian for detecting off-path packet injection vulnerabilities [9], and Zou *et al.* developed TCP-Fuzz, which systematically generates dependency-aware syscall and packet sequences to uncover inconsistencies between TCP stack implementations [44].

10 Conclusion

In this work, we introduced WAVED, the first systematic, automatic tool to identify off-path exploitable weak-verification vulnerabilities within the TCP/IP protocol suite. By involving a byte-level granularity taint analysis with direction-sensitive taint processing, WAVED can precisely model the constraints imposed by the input packet along any path to a specified risky operation, enabling it to filter and report paths with weak verification that may lead to semantic vulnerabilities. WAVED achieves an overall soundness with an MOP pointer analysis and taint analysis design, and the formatted user-friendly output design of WAVED helps users clearly figure out why and how the risky path is weakly constrained. We applied WAVED to the protocol stacks of three popular OS kernels, covering both IPv4 and IPv6 implementations, and uncovered 19 weak-verification vulnerabilities, 14 of which are newly found. WAVED can help developers better understand path constraints in complex protocol suite implementations, and be extended to support additional protocols and diverse forms of weak-verification vulnerabilities in the future.

Acknowledgment

We thank the anonymous reviewers and our shepherd for their insightful comments and constructive suggestions. We are also grateful to Qi Li for invaluable guidance and helpful feedback throughout this work. This work was supported in part by the National Science Foundation for Distinguished Young Scholars of China under Grant 62425201, the Science Fund for Creative Research Groups of the National Natural Science Foundation of China under Grant 62221003, and the Key Program of the National Natural Science Foundation of China under Grant 62132011. Xuewei Feng and Ke Xu are corresponding authors of this paper.

Ethical Considerations

Stakeholder Identification and Benefit-Risk Analysis. We identified two primary stakeholder groups impacted by this research: (1) End Users of the affected operating systems (Linux and FreeBSD); and (2) OS Maintainers and Developers.

- **End Users.** The primary risk to end users is the potential exploitation of the identified weak-verification vulnerabilities, which could lead to service disruptions (DoS) or traffic manipulation. However, the overarching benefit is the long-term improvement of the Internet infrastructure's security posture as these deep semantic flaws are remediated.
- **OS Maintainers and Developers.** Our research helps maintainers to review and patch complex logic bugs. To assist in remediation, we provided detailed proof-of-concept (PoC) exploits and, where possible, submitted patches (e.g., for the DCCP vulnerabilities). This collaborative approach helps the maintainers to efficiently understand the breaches we found.

Responsible Vulnerability Disclosure. Failing to disclose vulnerabilities responsibly could lead to serious security risks. We have established and adhered to a strict responsible disclosure process. All the vulnerabilities we found are responsibly reported to the Linux and FreeBSD security teams, providing them with sufficient time to reproduce and patch the issues before any public disclosure.

Mitigation of Dual-Use Risks. We recognize that the tools and techniques developed in this study have a dual-use nature and could potentially be misused by adversaries. To minimize this risk, this paper reports only the high-level conditions required to trigger vulnerabilities rather than providing weaponized, exploit-ready payloads. We restrict the release of full exploit chains, ensuring that the published material serves defensive purposes—helping developers understand and fix path constraints—rather than facilitating attacks.

Experimental Isolation and Safety. To avoid any negative impact on live systems, all our experiments were conducted strictly within isolated, offline lab environments (single-host setups and private virtual networks). No scanning, probing, or interaction with live, production networks was performed, ensuring no disruption to real-world services.

Post-Facto Ethical Analysis and Reflection. We explicitly acknowledge that the stakeholder and risk analysis presented in this section was conducted post-facto. While we did not perform a formal ethical framework analysis prior to the study, our experimental design inherently prioritized safety by operating strictly within isolated, offline environments. Reflecting on this process, had we performed a formal a priori analysis, our core technical methodology would likely have remained unchanged as it effectively minimized risk to live systems. However, an earlier formal analysis might have prompted us to engage with the ethical implications of "dual-use" tools sooner, potentially formalizing our disclosure timeline and risk warnings at an earlier stage.

Reasonableness of Publication. While we recognize that reporting and disclosing vulnerabilities carries inherent risks regarding legal liability and potential vendor retaliation, we believe that suppressing these findings would pose a greater threat to the security community. Public disclosure is reasonable and necessary to empower end-users to assess their risk exposure and to motivate vendors to implement timely fixes.

Guidance for Future Research. We recommend that future researchers in vulnerability discovery conduct a formal stakeholder analysis at the project's inception. Specifically, identifying stakeholders early on can help structure the research to maximize defensive utility while proactively minimizing the window of exposure during the disclosure phase.

Open Science

WAVED is open-sourced to facilitate reproducibility and artifact evaluation. We also provide guidance to help users reproduce results. The kernel source code of Linux and FreeBSD used in our experiments can be found at https://github.com/torvalds/linux and https://github.com/freebsd/freebsd-src. No extra datasets are involved. All the other necessary materials of WAVED including the source code and the user guideline are provided at https://doi.org/10.5281/zenodo.17896120.

References

[1] Fred Baker. Requirements for IP Version 4 Routers. RFC 1812, Internet Engineering Task Force, June 1995.

[2] Steven M Bellovin. Security problems in the tcp/ip protocol suite. *ACM SIGCOMM Computer Communication Review*, 19(2):32–48, 1989.

[3] Steven M Bellovin. A look back at "security problems in the tcp/ip protocol suite". In *20th Annual Computer Security Applications Conference*, pages 229–249. IEEE, 2004.

[4] Markus Brandt, Tianxiang Dai, Amit Klein, Haya Shulman, and Michael Waidner. Domain validation++ for mitm-resilient pki. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2060–2076, 2018.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[6] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path tcp exploits: Global rate limit considered dangerous. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 209–225, 2016.

[7] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path tcp exploits of the challenge ack global rate limit. *IEEE/ACM Transactions on Networking*, 26(2):765–778, 2018.

[8] Yue Cao, Zhongjie Wang, Zhiyun Qian, Chengyu Song, Srikanth V Krishnamurthy, and Paul Yu. Principled unearthing of tcp side channel vulnerabilities. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 211–224, 2019.

[9] Qi Alfred Chen, Zhiyun Qian, Yunhan Jack Jia, Yuru Shao, and Zhuoqing Morley Mao. Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 388–400, 2015.

[10] Weiteng Chen and Zhiyun Qian. Off-path tcp exploit: How wireless routers can jeopardize your secrets. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1581–1598, 2018.

[11] Douglas E Comer. *Internetworking with TCP/IP: principles, protocols, and architecture*. Prentice Hall, 1995.

[12] Alex Conta and Stephen Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443, Internet Engineering Task Force, March 2006.

[13] Wikipedia contributors. 64-bit computing. https://en.wikipedia.org/wiki/LP64, Accessed July 2025.

[14] Nmap Developers. Nmap github repository. https://github.com/nmap/nmap, 2025. Accessed December 2025.

[15] Xuewei Feng, Chuanpu Fu, Qi Li, Kun Sun, and Ke Xu. Off-path tcp exploits of the mixed ipid assignment. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 1323–1335, 2020.

[16] Xuewei Feng, Qi Li, Kun Sun, Chuanpu Fu, and Ke Xu. Off-path tcp hijacking attacks via the side channel of downgraded ipid. *IEEE/ACM transactions on networking*, 30(1):409–422, 2021.

[17] Xuewei Feng, Qi Li, Kun Sun, Zhiyun Qian, Gang Zhao, Xiaohui Kuang, Chuanpu Fu, and Ke Xu. Off-path network traffic manipulation via revitalized icmp redirect attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2619–2636, 2022.

[18] Xuewei Feng, Qi Li, Kun Sun, Ke Xu, Baojun Liu, Xiaofeng Zheng, Qiushi Yang, Haixin Duan, and Zhiyun Qian. Pmtud is not panacea: Revisiting ip fragmentation attacks against tcp. In *NDSS*, 2022.

[19] Xuewei Feng, Qi Li, Kun Sun, Ke Xu, and Jianping Wu. Exploiting cross-layer vulnerabilities: Off-path attacks on the tcp/ip protocol suite. In *Communications of the ACM (CACM)*, 2025.

[20] Xuewei Feng, Qi Li, Kun Sun, Yuxiang Yang, and Ke Xu. Man-in-the-middle attacks without rogue ap: When wpas meet icmp redirects. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3162–3177. IEEE, 2023.

[21] Xuewei Feng, Yuxiang Yang, Qi Li, Xingxiang Zhan, Kun Sun, Ziqiang Wang, Ao Wang, Ganqiu Du, and Ke Xu. Redan: An empirical study on remote dos attacks against nat networks. In *Network and Distributed System Security Symposium (NDSS)*, 2025.

[22] Yossi Gilad and Amir Herzberg. Off-path attacking the web. In *WOOT*, pages 41–52, 2012.

[23] Yossi Gilad and Amir Herzberg. Spying in the dark: Tcp and tor traffic analysis. In *International symposium on privacy enhancing technologies symposium*, pages 100–119. Springer, 2012.

[24] Yossi Gilad, Amir Herzberg, and Haya Shulman. Off-path hacking: The illusion of challenge-response authentication. *IEEE Security & Privacy*, 12(5):68–77, 2013.

[25] Eddie Kohler, Mark Handley, and Sally Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340, Internet Engineering Task Force, March 2006.

[26] Christopher Arthur Lattner. The llvm compiler infrastructure. https://llvm.org/, Accessed December 2024.

[27] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024, 2017.

[28] Keyu Man, Zhiyun Qian, Zhongjie Wang, Xiaofeng Zheng, Youjun Huang, and Haixin Duan. Dns cache poisoning attack reloaded: Revolutions with side channels.

In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1337–1350, 2020.

[29] Keyu Man, Zhongjie Wang, Yu Hao, Shenghan Zheng, Xin'an Zhou, Yue Cao, and Zhiyun Qian. Scad: Towards a universal and automated network side-channel vulnerability detection. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1861–1876. IEEE, 2025.

[30] Keyu Man, Xin'an Zhou, and Zhiyun Qian. Dns cache poisoning attack: Resurrections with side channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3400–3414, 2021.

[31] J McCann, S Deering, and J Mogul. Path mtu discovery for ip version 6. RFC 8201, Internet Engineering Task Force, July 2017.

[32] Jack McCann, Steve Deering, and Jeffrey Mogul. Path mtu discovery for ip version 6. RFC 1981, Internet Engineering Task Force, August 1996.

[33] Jeffrey Mogul and Steve Deering. Path mtu discovery. RFC 1191, Internet Engineering Task Force, November 1990.

[34] Jon Postel. Internet Control Message Protocol. RFC 792, Internet Engineering Task Force, September 1981.

[35] Zhiyun Qian and Z Morley Mao. Off-path tcp sequence number inference attack how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361. IEEE, 2012.

[36] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[37] Kaiwen Shen, Jianyu Lu, Yaru Yang, Jianjun Chen, Mingming Zhang, Haixin Duan, Jia Zhang, and Xiaofeng Zheng. Hdiff: A semi-automatic framework for discovering semantic gap attack in http implementations. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13. IEEE, 2022.

[38] Yulei Sui et al. Svf: Static value-flow analysis framework: commit a68b293. https://github.com/SVF-tools/SVF/commit/a68b29388634f6f2c88751ffcd3d500f10353930, October 2024. Accessed October 2024.

[39] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[40] The ZMap Team. The zmap project. https://zmap.io, 2025. Accessed December 2025.

[41] Yuxiang Yang, Xuewei Feng, Qi Li, Kun Sun, Ziqiang Wang, and Ke Xu. Exploiting sequence number leakage: Tcp hijacking in nat-enabled wi-fi networks. 2024.

[42] Feiyang Yu, Quan Zhou, Syed Rafiul Hussain, and Danfeng Zhang. Athena: analyzing and quantifying side channels of transport layer protocols. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3117–3133, 2024.

[43] Xiaofeng Zheng, Chaoyi Lu, Jian Peng, Qiushi Yang, Dongjie Zhou, Baojun Liu, Keyu Man, Shuang Hao, Haixin Duan, and Zhiyun Qian. Poison over troubled forwarders: A cache poisoning attack targeting dns forwarding devices. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 577–593, 2020.

[44] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. Tcp-fuzz: Detecting memory and semantic bugs in {TCP} stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 489–502, 2021.

Appendix A. Algorithms

Algorithm 1: Construct Context-Sensitive SVFG

Input: The context-insensitive call graph G_{call}
The original Sparse Value-Flow Graph G_{svf}
Output: Context-Sensitive SVFG G_{cssvf}

Function `buildCSSVFG()`:
- worklist $\leftarrow (fun_{entry}, \varepsilon)$, ($\varepsilon$ is the empty context);
- $\mathcal{C}(fun_{entry}) \leftarrow \{\varepsilon\}$;
- **while** *worklist is not empty* **do**
  - Pop (fun, ctx) from worklist;
  - **foreach** $fun \rightarrow fun'$ *in* G_{call} **do**
    - $ctx' \leftarrow$ `computeCtx(`$ctx, fun \rightarrow fun'$`)`;
    - **if** $ctx' \notin \mathcal{C}(fun')$ **then**
      - $\mathcal{C}(fun') \leftarrow \mathcal{C}(fun') \cup \{ctx'\}$;
      - Append (fun', ctx') to worklist;
- **foreach** node $u \in G_{svf}$ **do**
  - **foreach** ctx *in* $\mathcal{C}(fun(u))$ **do**
    - Add node (ctx, u) into G_{cssvf};
- **foreach** edge $u \rightarrow v \in G_{svf}$ **do**
  - **foreach** ctx *in* $\mathcal{C}(fun(u))$ **do**
    - **if** $fun(u) = fun(v)$ **then**
      - Add $(ctx, u) \rightarrow (ctx, v)$ into G_{cssvf};
    - **else**
      - Calculate ctx' based on ctx and $u \rightarrow v$;
      - Add $(ctx, u) \rightarrow (ctx', v)$ into G_{cssvf};
- **return** G_{cssvf};

Algorithm 2: Byte-level Taint Propagation via Multiplication

Input: Two taint propagation value $\mathcal{M}_{u\rightarrow v}$ and $\mathcal{M}_{v\rightarrow w}$ to multiply
Output: The result taint propagation value $\mathcal{M}_{u\rightarrow w}$

Function `init_mask()`:
- **for** $i \leftarrow 0$ **to** $2^8 - 1$ **do**
  - $mask \leftarrow 0$;
  - **for** $j \leftarrow 0$ **to** 7 **do**
    - $bit \leftarrow ((i \gg j) \,\&\, 1)$;
    - $mask \leftarrow mask \mid (bit \ll (j \cdot 8))$;
  - `Mask`$[i] \leftarrow mask$;
  - $mask_{row} \leftarrow 0xFF$;
  - $mask_{col} \leftarrow 0x101010101010101$;

Function `mult(`$\mathcal{M}_{u\rightarrow v}$, $\mathcal{M}_{v\rightarrow w}$`)`:
- $\mathcal{M}_{u\rightarrow w} \leftarrow 0$;
- **for** $i \leftarrow 0$ **to** 7 **do**
  - $row[i] \leftarrow (\mathcal{M}_{u\rightarrow v} \gg (i \cdot 8))$ & $mask_{row}$;
  - $col[i] \leftarrow (\mathcal{M}_{v\rightarrow w} \gg i)$ & $mask_{col}$;
- **for** $i \leftarrow 0$ **to** 7 **do**
  - **for** $j \leftarrow 0$ **to** 7 **do**
    - $\mathcal{M}_{u\rightarrow w} \leftarrow \mathcal{M}_{u\rightarrow w} \ll 1$;
    - **if** $(\texttt{Mask}[row[i]] \,\&\, col[j]) \neq 0$ **then**
      - $\mathcal{M}_{u\rightarrow w} \leftarrow \mathcal{M}_{u\rightarrow w} \mid 1$;
- return $\mathcal{M}_{u\rightarrow w}$;

Algorithm 3: Calculate Function Summary

Input: Context ctx, Function fun, Taint Propagation Graph G_{tpg}
Output: `Function Summaries` $\mathcal{S}(ctx, fun, ...)$

Function `ComputeSummary(`F, G_{tpg}`)`:
- $\mathcal{S}(ctx, fun, entry_{fun}) \leftarrow \{(\emptyset, \langle\phi, \phi, \phi\rangle)\}$
- **foreach** basic block u in topological order **do**
  - $term \leftarrow$ terminator instruction of u;
  - **foreach** $(\mathcal{P}_i, \mathcal{B}_{path}(\mathcal{P}_i)) \in \mathcal{S}(ctx, fun, u)$ **do**
    - **foreach** basic block $v \in u.successors$ **do**
      - $\mathcal{P}' \leftarrow$ control-dependent transitions to v in $\mathcal{P}_i$;
      - $\mathcal{P}_{new} \leftarrow \mathcal{P}' \cup (u, edge)$;
      - $\mathcal{C}_{edge} \leftarrow \langle\phi, \phi, \phi\rangle$;
      - **if** `IsCondBr(`$term$`)` **then**
        - $\langle\mathcal{B}_1, \mathcal{B}_0\rangle \leftarrow$ `ResolveBranchImpact(`$term, G_{tpg}$`)`;
        - **if** $isTrueEdge(u \rightarrow v)$ **then**
          - $\mathcal{C}_{edge} \leftarrow \mathcal{B}_1$;
        - **else**
          - $\mathcal{C}_{edge} \leftarrow \mathcal{B}_0$;
      - $\mathcal{B}_{new} \leftarrow \mathcal{B}_{path}(\mathcal{P}') \wedge \mathcal{C}_{edge}$;
      - **if** $(\mathcal{P}_{new}, \mathcal{B}_{new}) \notin \mathcal{S}(ctx, fun, v)$ **then**
        - $\mathcal{S}(ctx, fun, v).add((\mathcal{P}_{new}, \mathcal{B}_{new}))$;
- **return** $\mathcal{S}(ctx, fun, ...)$;

Algorithm 4: Construct Summary Graph

Input: `Function Summaries` $\mathcal{S}(ctx, fun, \dots)$
The context-sensitive call graph G_{cscall}
Output: Summary Graph G_{summ}

Function `buildSummaryGraph()`:
- **foreach** node $(ctx, fun) \in G_{cscall}$ **do**
  - Add node (ctx, fun) into G_{summ};
- **foreach** node $(ctx, fun) \in G_{summ}$ **do**
  - **foreach** path $\mathcal{P} \in \mathcal{S}(ctx, fun, \dots)$ **do**
    - $bb \leftarrow$ destination of $\mathcal{P}$;
    - **foreach** call site $cs \in bb$ **do**
      - **foreach** callee fun' of cs **do**
        - $ctx' \leftarrow$ `computeCtx(`ctx, cs`)`;
        - Add $(ctx, fun) \rightarrow (ctx', fun')$ into G_{summ} with weight $\mathcal{P}$;
- **return** G_{summ};