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

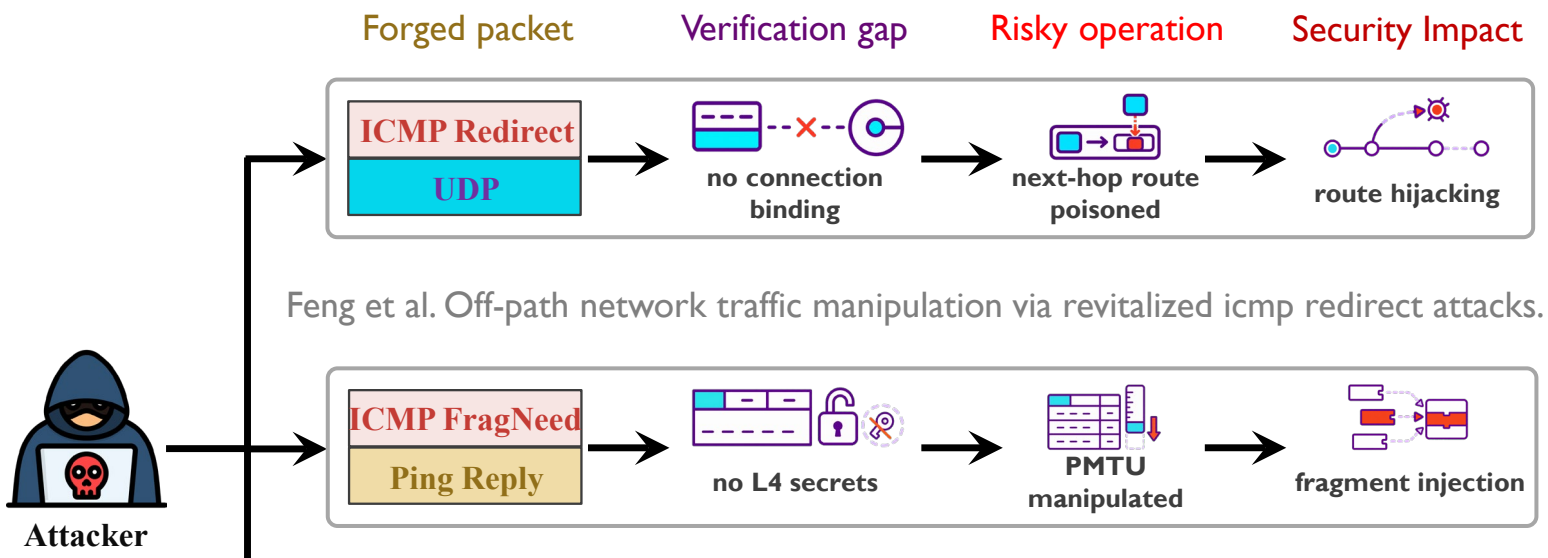
Yizhou Zhao [1)], Xuewei Feng [1)], Min Li [2)], Ke Xu [1) 2)]

1) *Tsinghua University*
2) *Zhongguancun Laboratory*





Off-path attacks repeatedly exploit verification gaps within the TCP/IP suite



Feng et al. Off-path network traffic manipulation via revitalized icmp redirect attacks. In USENIX Security'22.

Feng et al. Pmtud is not panacea: Revisiting ip fragmentation attacks against tcp. In NDSS'22.

Man et al. Dns cache poisoning attack: Resurrections with side channels. In ACM CCS'21.

An illustrative example

① Attacker forges an ICMP Fragment Needed quoting a ICMP Echo Reply

➞ Path MTU reduced, subsequent packets fragmented

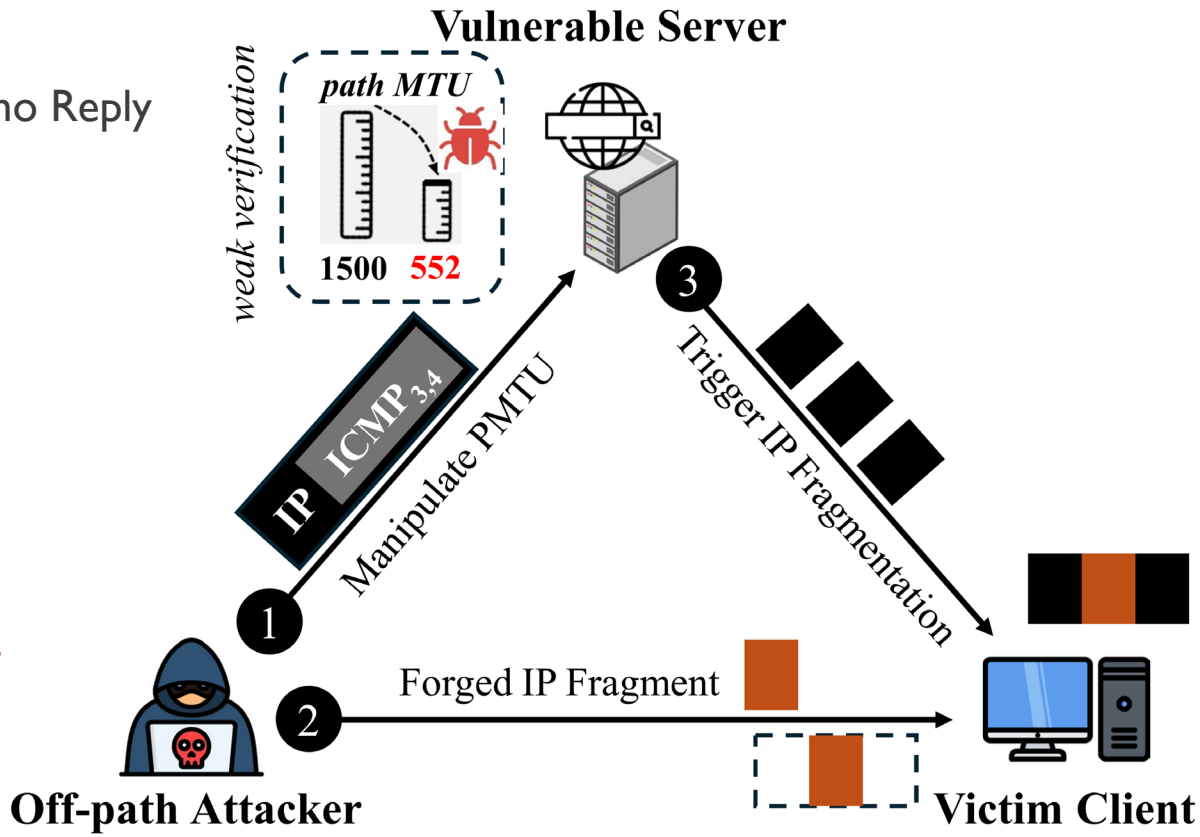
② Attacker forges an IP Fragment

➞ Received and cached by client

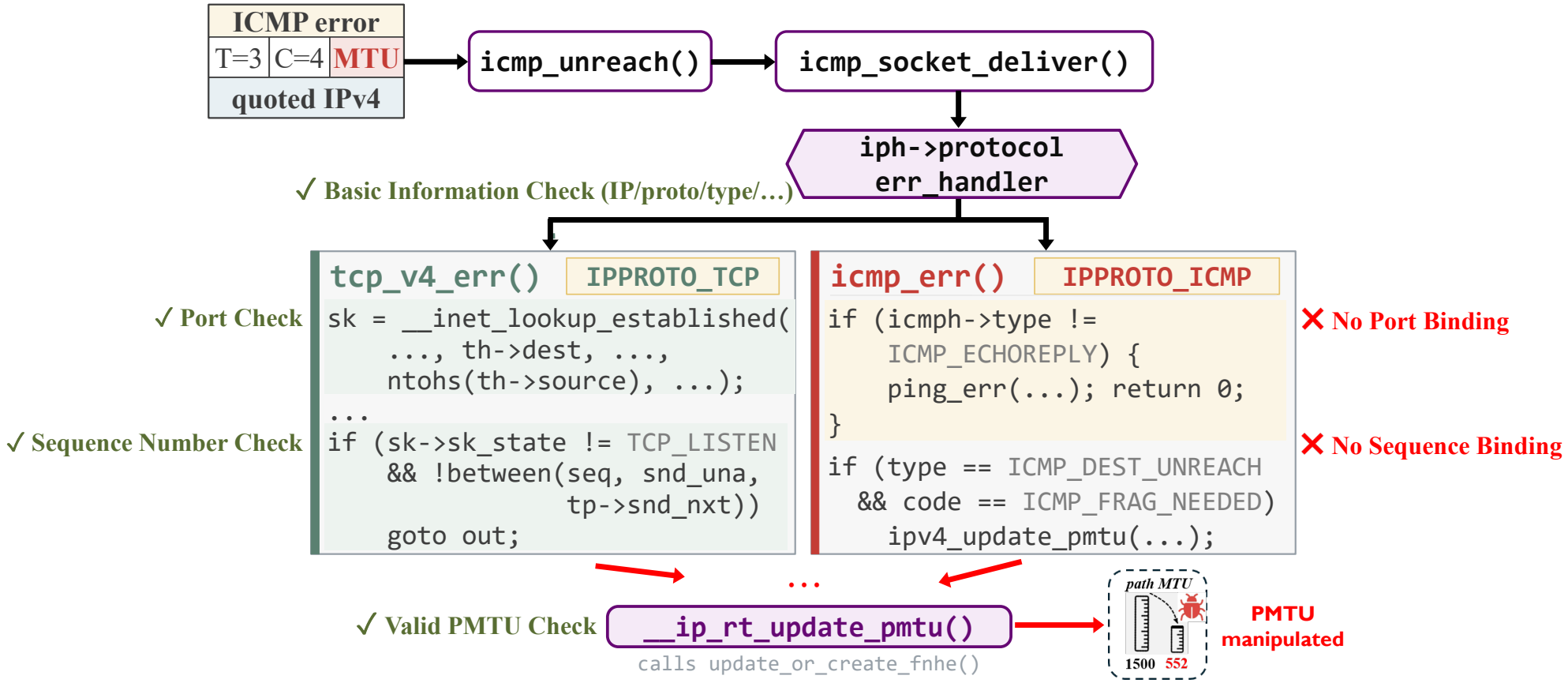
③ Server sends fragmented packets

➞ Assembled with malicious attacker-controlled fragment at client

➞ **Fragment Injection!**

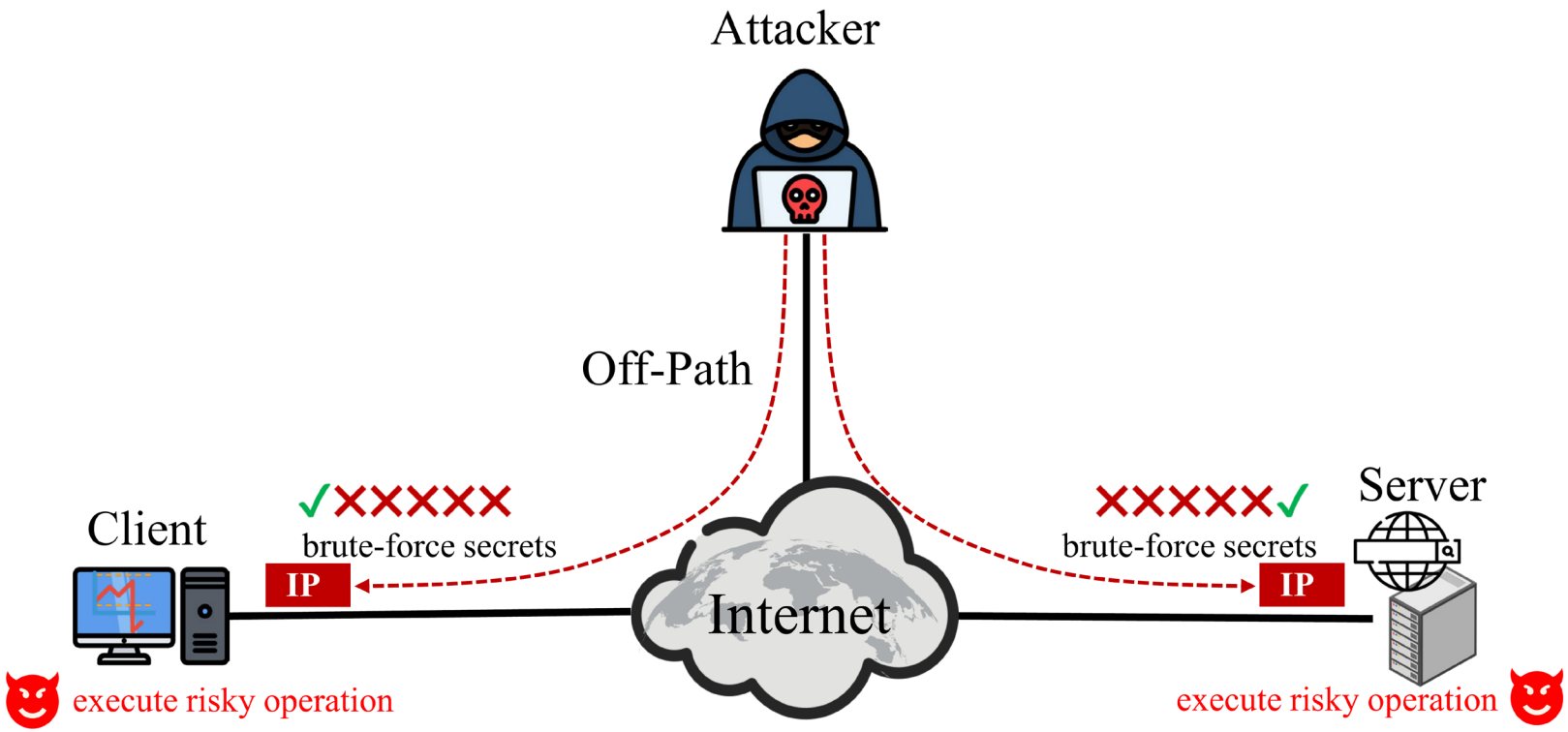


ICMP paths reach the same PMTU update with unequal checks



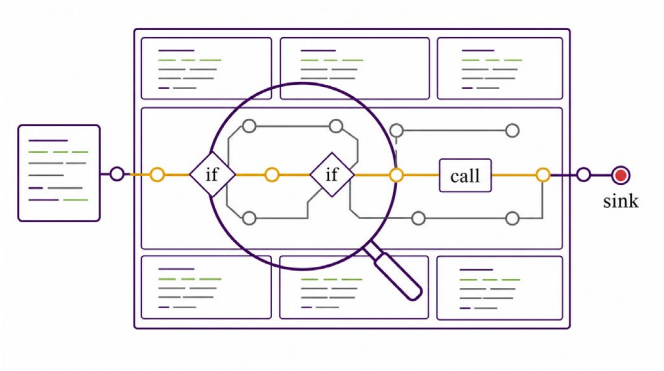
More checks are enforced on the TCP path than on the ICMP path, yet both reach the same state-changing PMTU-update sink

Threat model of attacks based on weak verifications



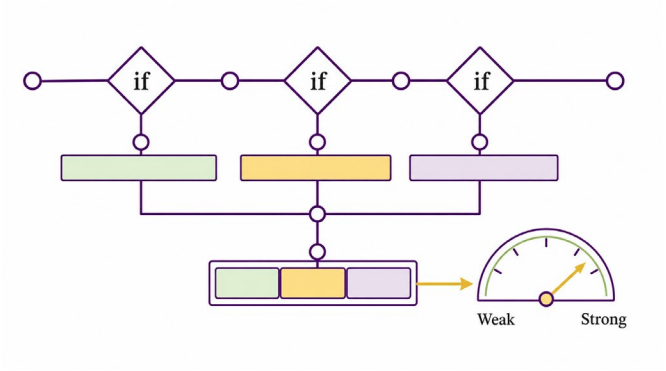
An off-path attacker cannot directly observe benign traffic, but can guess required secrets and spoof packets to trigger a risky operation in the protocol stack.

Scalable weak-verification discovery requires three capabilities



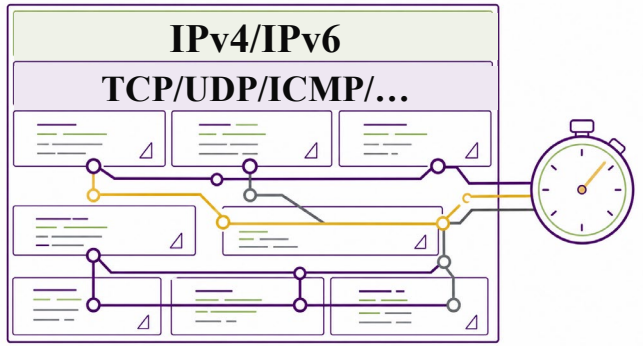
I. Precise real-code analysis

Follow actual execution paths and checks from kernel code.



II. Constraint composition & strength quantification

Quantify the strength of composed checks along paths.



III. Large-scale analysis efficiency

Analyze large kernel protocol codebases within practical time.

Limitations of existing approaches

Approach	Precise real-code analysis	Constraint composition & strength quantification	Large-scale analysis efficiency
Fuzzing e.g. [Zou 2021] TCP-Fuzz	△ real executions; triggered paths only	× test-case-specific	× costly with coverage gaps
Symbolic execution e.g. [Man 2025] SCAD	✓ precise explored paths	✓ explicit path conditions	× path explosion
Model checking e.g. [Cao 2019] SCENT	× abstracted model	△ model constraints	△ reduced scope
Static analysis e.g. [Chen 2015] PacketGuardian, [Yu 2024] Athena	✓ broad real code	× coarse/manual strength quantification	✓ scalable foundation

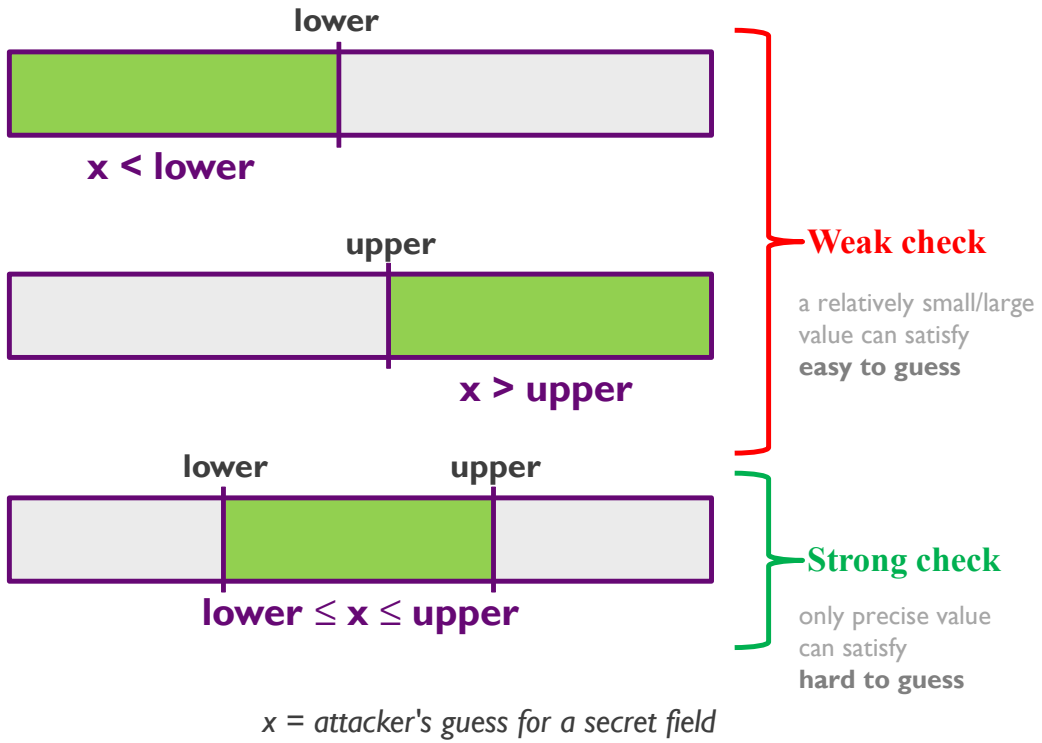
Our solution: Integrate constraint composition & strength quantification into static analysis

How to model path strengths inside the TCP/IP suite

Packet verification typically reduces to value constraints on specific packet fields

Not all constraints are equally difficult for an off-path attacker to satisfy, depending on the direction of the comparison

We model path strength by the attacker's guessing cost, measured by the number of strongly constrained packet bytes



Therefore, we introduce our tool, WAVED (WeAk Verification Exploitation Detector), integrating direction sensitivity into static analysis

Challenge Overview

Analysis precision

Distinguish between different caller contexts

```
ret = INDIRECT_CALL_2(ipprot->handler,
                      tcp_v4_rcv, udp_rcv, skb);
```



Problem: A context-insensitive analysis can cross-pair skb and length facts from different callers.

Preserve precision in pointer subtraction

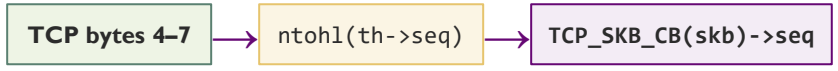
`field #n + (-k bytes) = ?`

field-index-based (e.g. SVF) ✗

byte-offset-based (ours) ✓

Problem: A field-index-based GEP resolution cannot consume a signed byte displacement; without field↔byte conversion, the enclosing object is lost.

Identify which original bytes reach the check

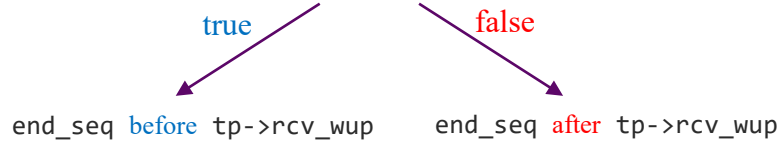


Problem: Object-level taint cannot identify original byte flow.

Constraint quantification

Differentiate T/F outcomes of the same branch

```
if (before(end_seq, tp->rcv_wup))
```



Problem: True and false outcomes of the same branch impose opposite directions, but a traditional taint analysis fails to recognize this.

Compose branch constraints along the path



Problem: How to design arithmetic rules in order to compose different branch constraints.

Recognize data-dependent implicit CF

```
reason = tcp_sequence(tp,
    TCP_SKB_CB(skb)->seq,
    TCP_SKB_CB(skb)->end_seq);
```

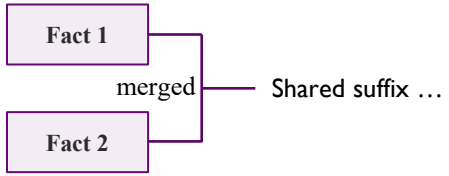


```
if (reason) {
    ...
    goto discard;
}
```

Problem: Data flow may implicitly encode downstream control-flow decisions that explicit-flow-only taint analysis can miss.

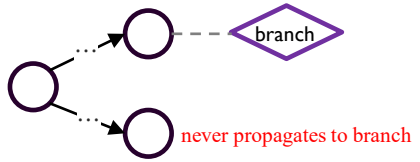
Large-scale analysis efficiency

Reuse merged data-flow facts



Problem: Path-by-path propagation repeats the same downstream analysis after paths converge, leading to path explosion.

Resolve constraint directions on demand



Problem: Eager analysis carries direction information through every value, even when that direction is never used as branch constraint, which reduces efficiency.

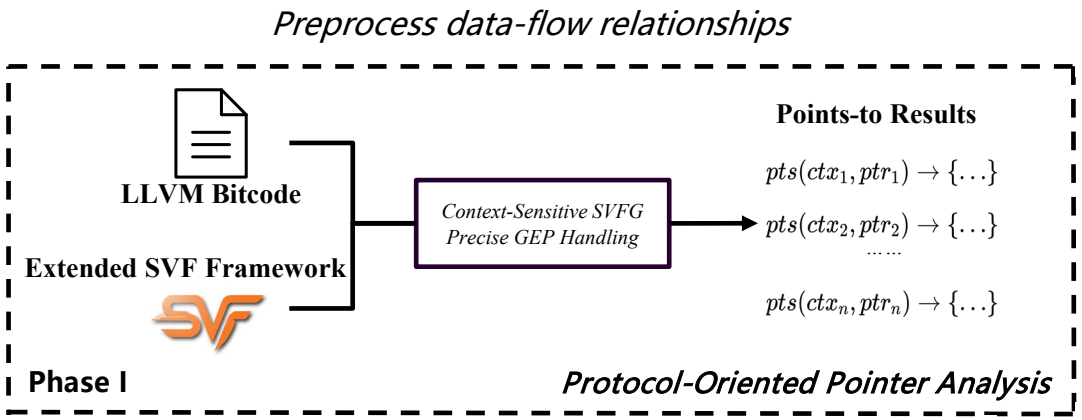
Cache function-local path constraints

$P_1: A \to B \to f$
$P_2: X \to Y \to f$

calculate constraints in function f twice

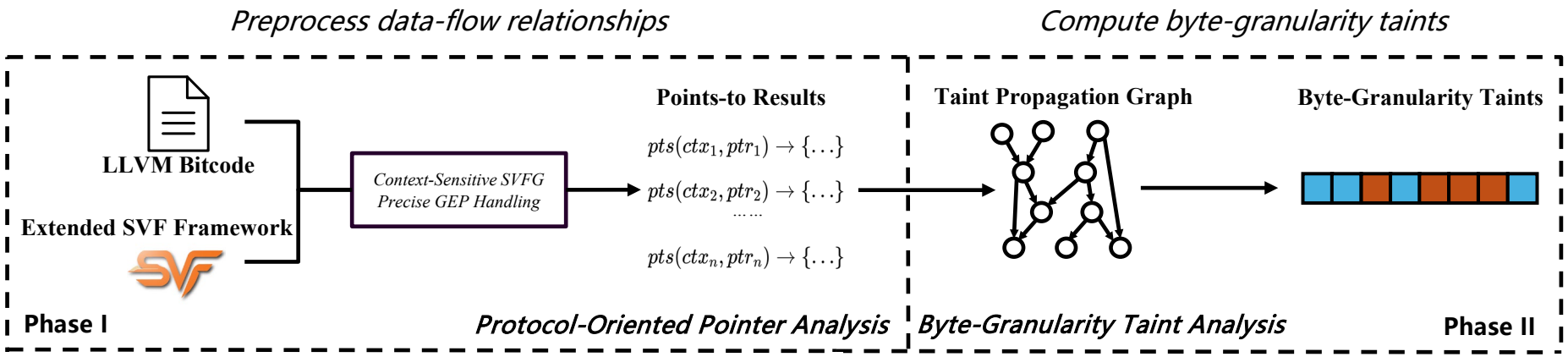
Problem: Different caller prefixes repeatedly recompute the same function-local path constraints.

Overview of **WAVED**



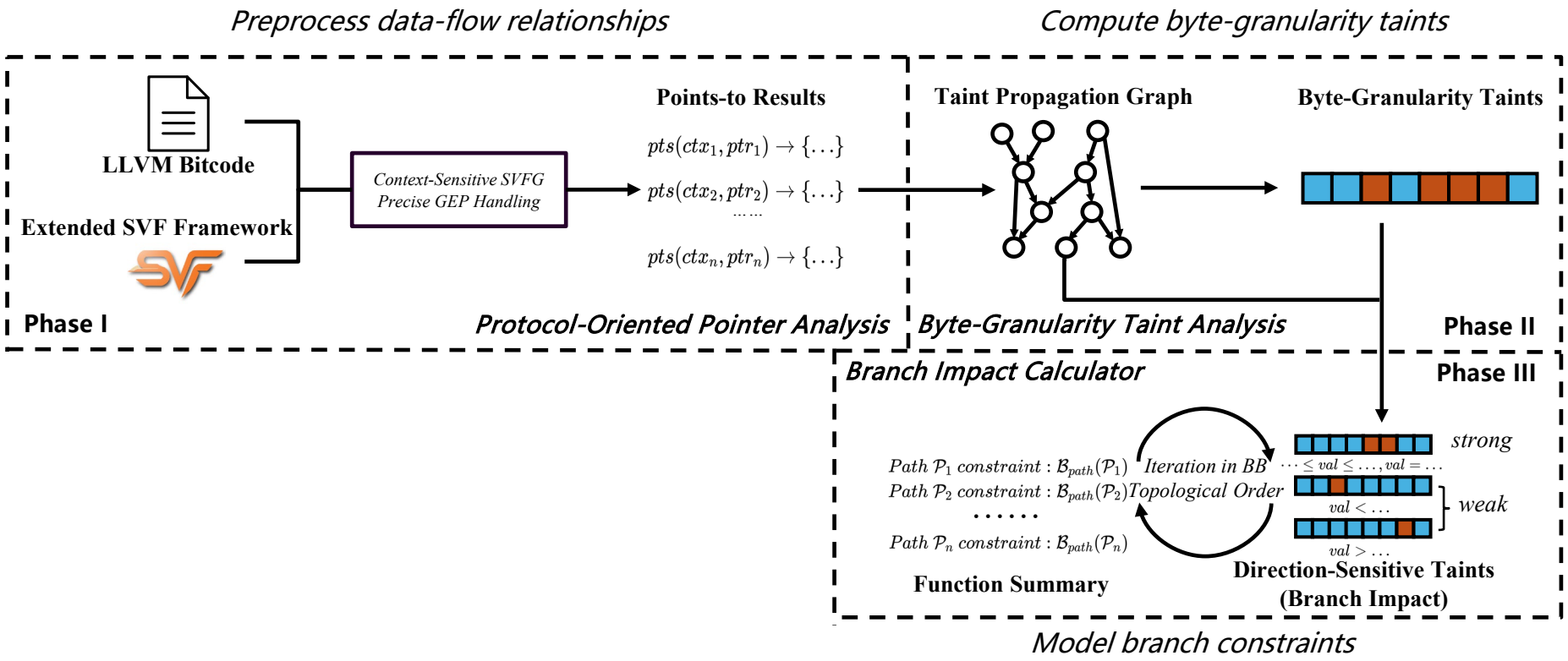
One end-to-end analysis, four composable phases.

Overview of **WAVED**



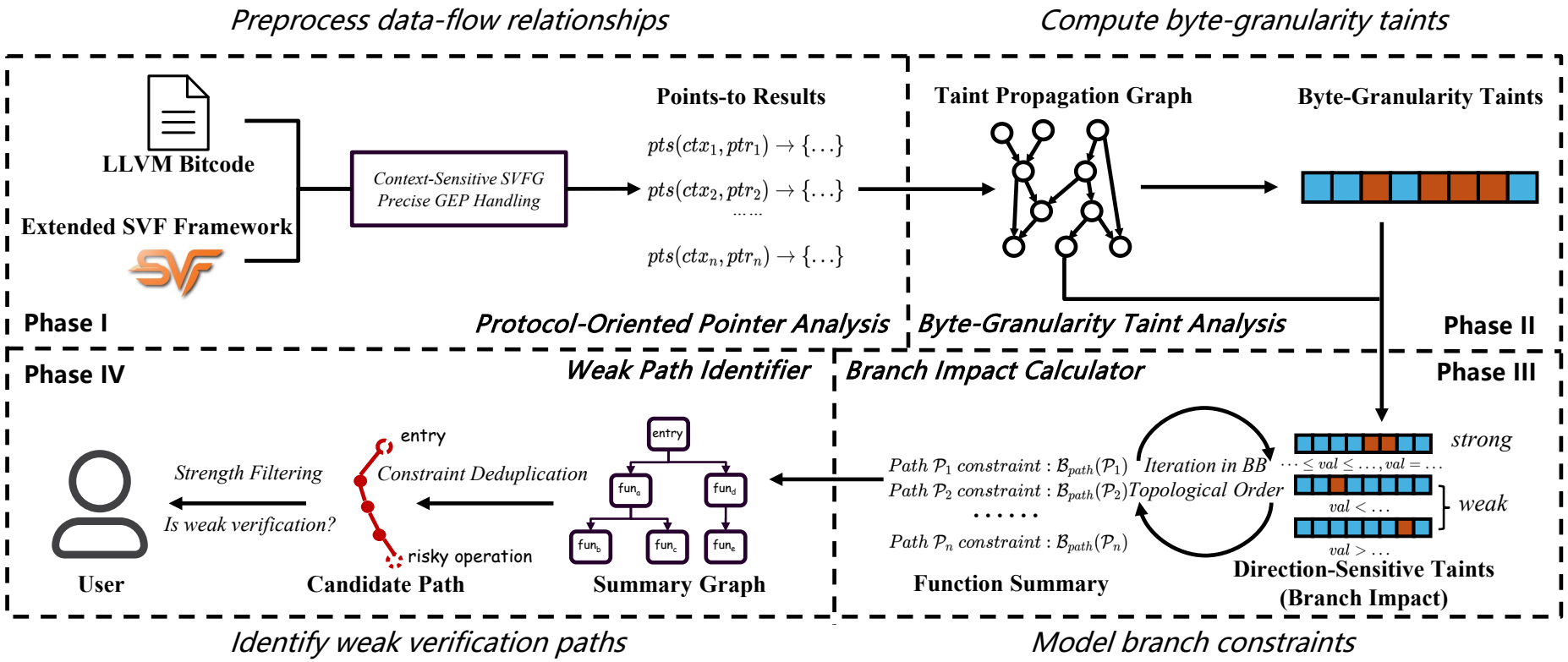
One end-to-end analysis, four composable phases.

Overview of **WAVED**



One end-to-end analysis, four composable phases.

Overview of **WAVED**

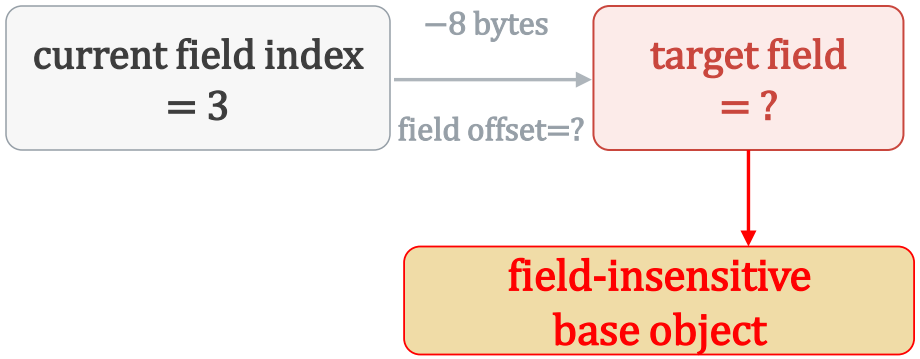


One end-to-end analysis, four composable phases.

Byte-offset-based GetElementPtr resolution

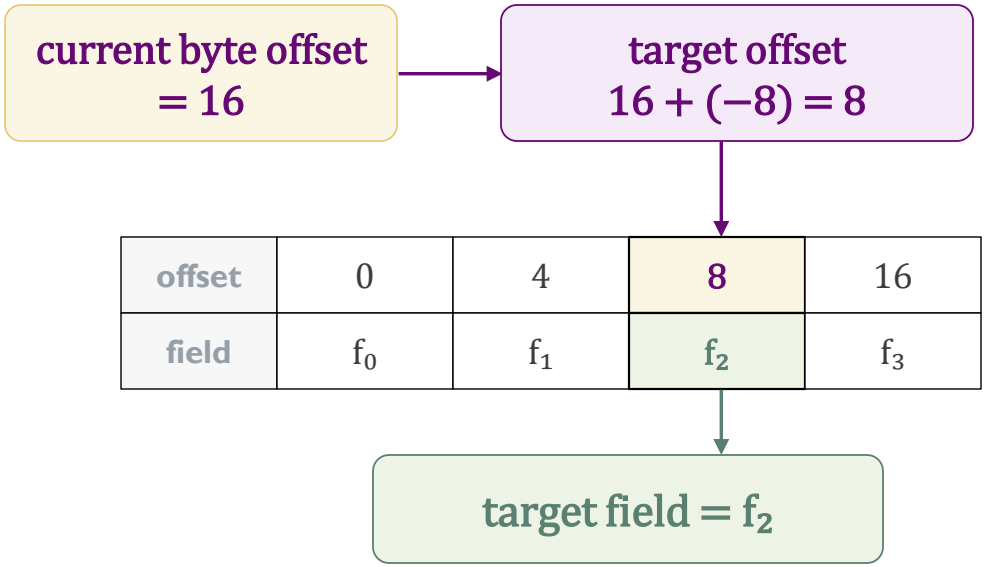
Input pointer: field f_3 at byte offset 16 • GEP displacement: $\Delta = -8$ bytes

SVF: flattened field index



Traditional pointer analysis implementations (e.g. in SVF) cannot interpret a negative byte displacement as a field offset, so the result degrades to a field-insensitive base object

WAVED: signed byte offsets



Keep the displacement in bytes; resolve the field after computing the target byte offset

Extending the Sparse Value-Flow Graph (SVFG) with context sensitivity

Context-sensitive SVFG nodes

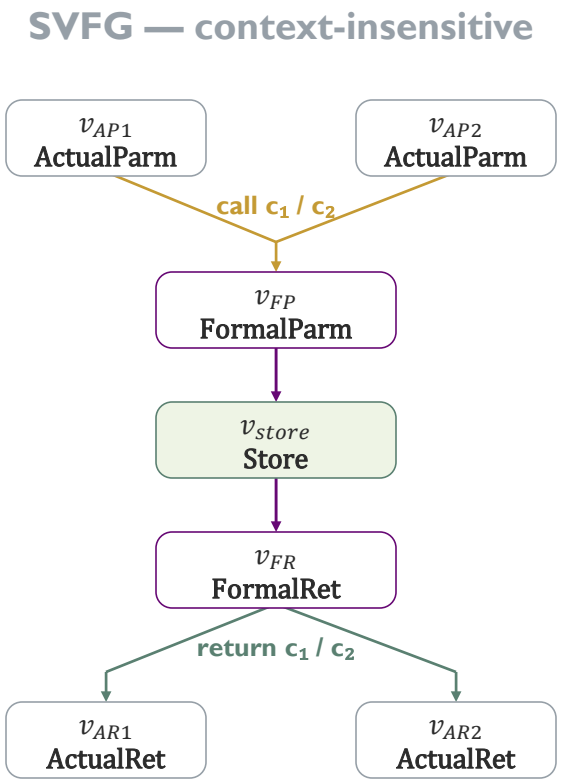
$\hat{v} = \langle \kappa, v \rangle, \quad v \in V_{SVFG}, \quad \kappa \in C\big(fun(v)\big), \quad |\kappa| \leq 3$

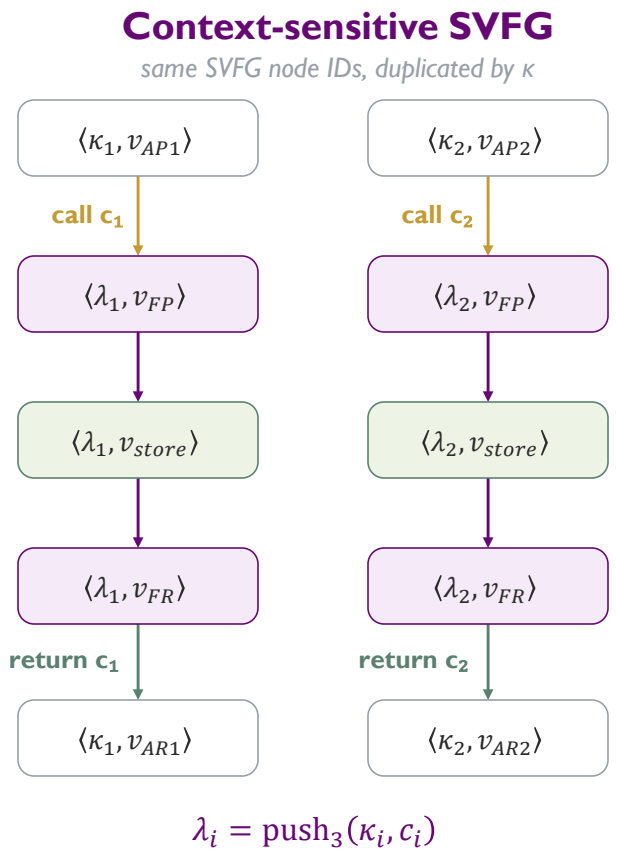
Context-sensitive SVFG edges

$intra$: $\langle \kappa, u \rangle \rightarrow \langle \kappa, v \rangle$

$call_{cs}$: $\langle \kappa, AP \rangle \rightarrow \langle push_{3(\kappa,cs)}, FP \rangle$

$return_{cs}$: $\langle push_{3(\kappa,cs)}, FR \rangle \rightarrow \langle \kappa, AR \rangle$





$\lambda_i = \mathrm{push}_3(\kappa_i, c_i)$

Context sensitivity distinguishes value flows across complex kernel calling contexts, improving interprocedural precision of the pointer analysis.

Locating byte-level data flow – Taint Propagation Graph

TPG extends the CSSVFG with byte-to-byte relations on each edge

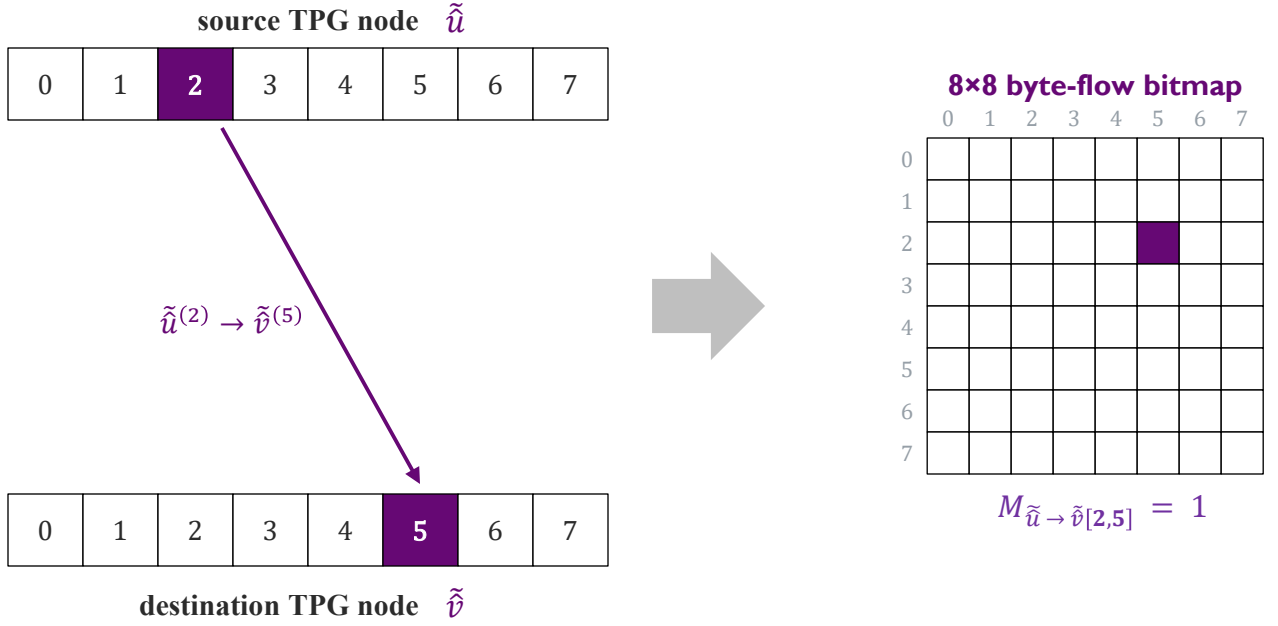
$$V_{TPG} = \{\tilde{v} \mid \hat{v} \in V_{CSSVFG}\}$$

$$E_{TPG} = \{\tilde{u} \to \tilde{v} \mid \hat{u} \to \hat{v} \in E_{CSSVFG}\}$$

Byte flows are preserved inside each TPG edge

$$Bytes(\tilde{v}) = \{\tilde{v}^{(i)} \mid 0 \le i < \ell(\tilde{v})\}, \qquad \ell(\tilde{v}) \le 8$$

$$M_{\tilde{u} \to \tilde{v}[i,j]} = 1 \iff \tilde{u}^{(i)} \rightsquigarrow \tilde{v}^{(j)}$$



TPG further extends context-sensitive data flow with byte-to-byte relations, representing the fine-grained byte-level taint flow.

Mapping LLVM IR instructions to seven transfer rules

Transfer rule	IR operation / relation	Bitmap	Mapping rule
Copy	Cast / Phi / Store / Select / Branch call-return / indirect value-flow/…		$M_{\widetilde{u}\to\tilde{v}[i,j]} = [i = j]$
And(c)	bitwise AND with constant c		$M_{\widetilde{u}\to\tilde{v}[i,j]} = [i = j \ \wedge \ c.byte[i] \neq 0]$
Or(c)	bitwise OR with constant c		$M_{\widetilde{u}\to\tilde{v}[i,j]} = [i = j \ \wedge \ c.byte[i] \neq 0xFF]$
LogicOff(o)	Shl / LShr with constant offset o (bits)		$M_{\widetilde{u}\to\tilde{v}[i,j]} = \left[\left\lfloor\frac{o}{8}\right\rfloor \leq j - i \leq \left\lceil\frac{o}{8}\right\rceil\right]$
ArithOff(o,s)	AShr with offset o (bits) and byte size s		$M_{\widetilde{u}\to\tilde{v}[i,j]} = \left[\left\lfloor\frac{o}{8}\right\rfloor \leq i - j \leq \left\lceil\frac{o}{8}\right\rceil \vee \left(i = s - 1 \wedge 0 \leq s - 1 - j < \left\lceil\frac{o+7}{8}\right\rceil\right)\right]$
LowToHigh	GEP(base) / Add / Sub / Mul		$M_{\widetilde{u}\to\tilde{v}[i,j]} = [i \leq j]$
Full	all remaining instructions / relations		$M_{\widetilde{u}\to\tilde{v}[i,j]} = 1$

Starting from every input-packet byte, WAVED forward-slices the TPG along Context-Free-Language(CFL)-valid call/return paths.

Composing the byte-flow relations along these paths preserve the exact Meet-Over-all-Paths(MOP) result within the TPG abstraction.

$$\text{Taint}(v) = \{\, k \mid \text{input packet byte } b[k] \text{ may influence } v \,\}$$

Taint tuples encode direction-sensitive byte constraints

DEFINITION 1 — TAINT TUPLE

$$\mathcal{T} = \langle L, G, E \rangle$$

***L*, *G*, *E* are sets of original input-packet byte offsets that may influence constraint *c*.**

- L - less-than / upper-bound direction
- G - greater-than / lower-bound direction
- E - equality or bounded range

Cmp	Taint Tuple
$a < b$ *or* $a \leq b$	$\langle \text{Taint}(a), \text{Taint}(b), \phi \rangle$
$a > b$ *or* $a \geq b$	$\langle \text{Taint}(b), \text{Taint}(a), \phi \rangle$
$a = b$	$\langle \phi, \phi, \text{Taint}(a) \cup \text{Taint}(b) \rangle$
$a \neq b$	$\langle \phi, \phi, \phi \rangle$

Define ϕ as the empty set.

$$\mathcal{T}_1 = \langle L_1, G_1, E_1 \rangle \qquad \mathcal{T}_2 = \langle L_2, G_2, E_2 \rangle$$

$$\mathcal{T}_1 \wedge \mathcal{T}_2 = \langle L_1 \cup L_2, G_1 \cup G_2, E_1 \cup E_2 \rangle$$

$$\mathcal{T}_1 \vee \mathcal{T}_2 = \langle L_1, G_1, E_1 \rangle \vee \langle L_2, G_2, E_2 \rangle$$

$x < \text{upper}$, $k \in L$ $\wedge$ $x > \text{lower}$, $k \in G$ $=$ $\text{lower} < x < \text{upper}$, $k \in E$

s.t. $k \in \text{Taint}(x)$

$$\langle L, G, E \rangle = \langle L \setminus (L \cap G), G \setminus (L \cap G), E \cup (L \cap G) \rangle$$

Branch impacts preserve every alternative constraint under each value

DEFINITION 2 — BRANCH IMPACT

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

A branch impact $\mathcal{B}$ is the disjunction of every feasible taint tuple $\mathcal{T}^{(i)}$ (i.e. every possible constraint) under each value.

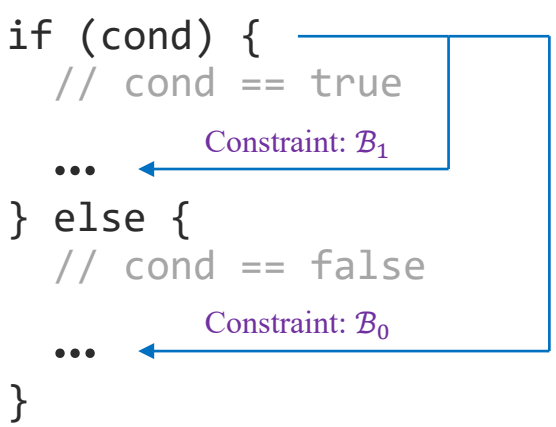
$$\mathcal{B}_1 = \bigvee_{i=1}^{m} \mathcal{T}_1^{(i)} \qquad \mathcal{B}_2 = \bigvee_{j=1}^{n} \mathcal{T}_2^{(j)}$$

$$\mathcal{B}_1 \wedge \mathcal{B}_2 = \bigvee_{i=1}^{m} \bigvee_{j=1}^{n} \left(\mathcal{T}_1^{(i)} \wedge \mathcal{T}_2^{(j)} \right)$$

$$\mathcal{B}_1 \vee \mathcal{B}_2 = \left(\bigvee_{i=1}^{m} \mathcal{T}_1^{(i)} \right) \vee \left(\bigvee_{j=1}^{n} \mathcal{T}_2^{(j)} \right)$$

Track separate branch impacts for the true and false outcomes of each branch

Maintain a pair of branch impacts
$\langle \mathcal{B}_1, \mathcal{B}_0 \rangle = \langle \text{true outcome}, \text{false outcome} \rangle$
at each conditional branch.



Demand-driven branch impact resolution

ALGORITHM — RECURSIVE, DEMAND-DRIVEN RESOLUTION returns $\langle \mathcal{B}_1, \mathcal{B}_0 \rangle$

```
1  CollectBranchImpact(n):
2      if visited[n]: return impact[n]
3      visited[n] ← true
4      if n is a direct comparison:
5          impact[n] ← ⟨B₁, B₀⟩ from its predicate and operand taints
6      else:
7          D ← RequiredDefinitions(n)
8          for d ∈ D:
9              impact[d] ← CollectBranchImpact(d)
10     impact[n] ← ResolveBySemantics(n, {impact[d] | d ∈ D})
11         ...
12     return impact[n]
```

Resolve branch impacts backward and on demand, rather than eagerly resolving constraint directions for every value.

Using instruction-specific semantics for different LLVM instructions (e.g. Cmp, Binary, Unary, Phi, Select, Cast, Call, …).

Deriving implicit control flow constraints from value flow

Linux 6.8 ·tcp_input.c

helper ·tcp_sequence()

```
static enum skb_drop_reason tcp_sequence(...) {
  if (before(end_seq, tp->rcv_wup))
    return SKB_DROP_REASON_TCP_OLD_SEQUENCE; // 33

  if (after(seq, tp->rcv_nxt +
            tcp_receive_window(tp)))
    return SKB_DROP_REASON_TCP_INVALID_SEQUENCE; // 34

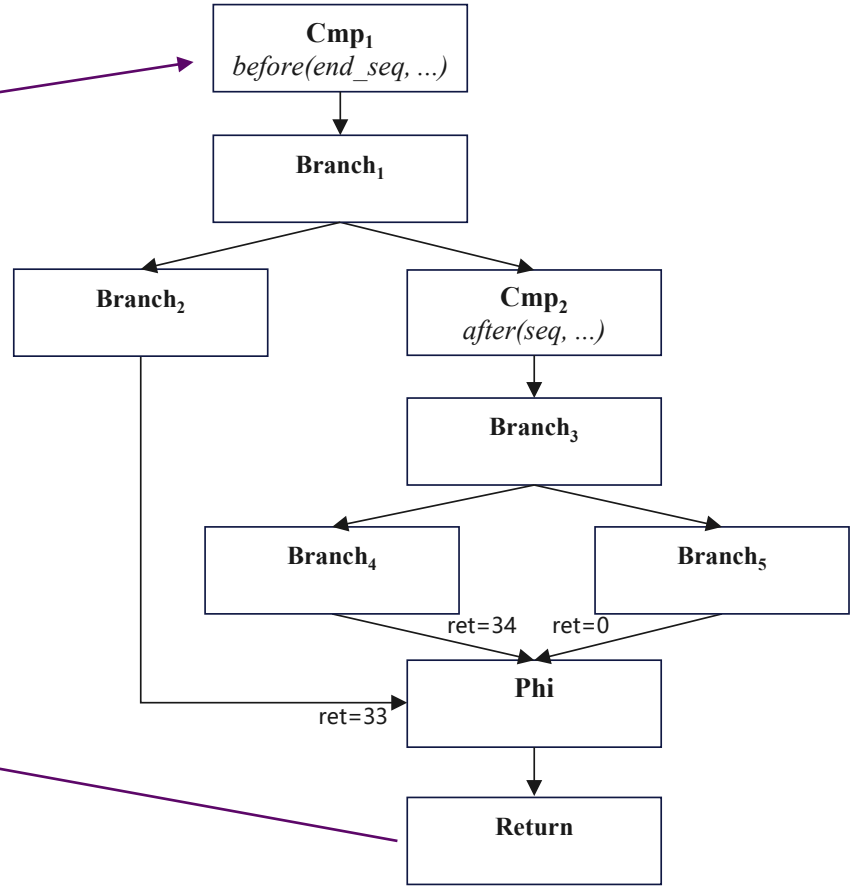
  return SKB_NOT_DROPPED_YET; // 0
}
```

caller ·tcp_validate_incoming()

```
reason = tcp_sequence(tp,
    TCP_SKB_CB(skb)->seq,
    TCP_SKB_CB(skb)->end_seq);

if (reason) {
    ...
    goto discard;
}
```

The returned constants further determine the caller's control flow.



Deriving implicit control flow constraints from value flow

Compose control-dependent branch impacts into path constraints during CFG traversal.

Merge path states that reach the same basic block with identical control-dependent branch-decision histories to improve efficiency.

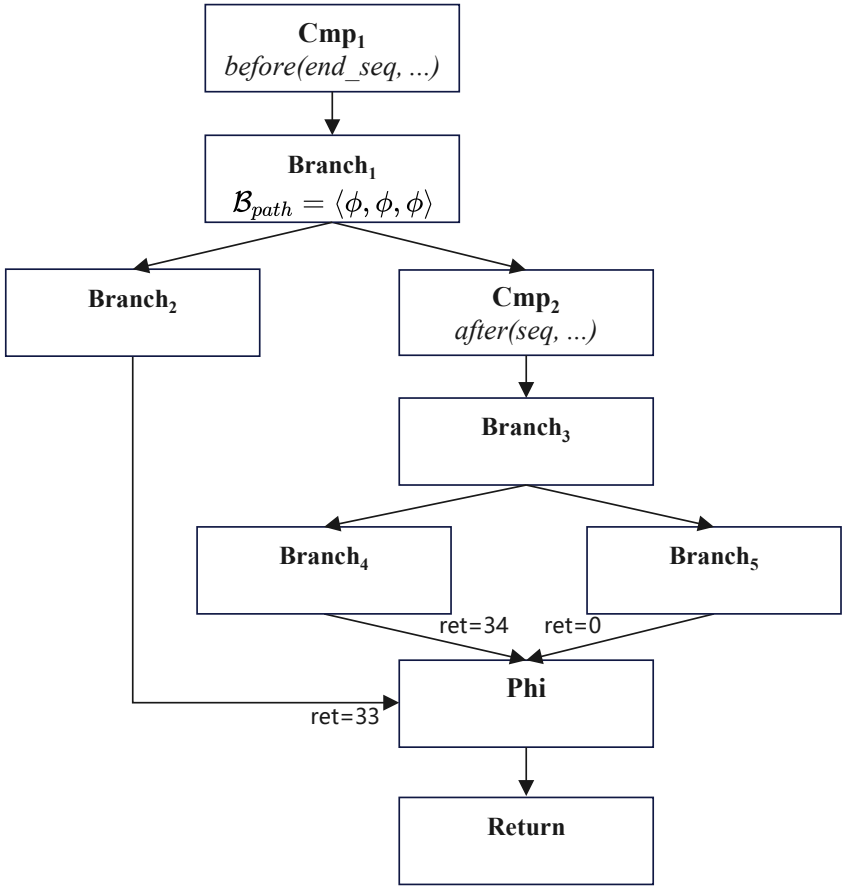
Meanwhile, associate each value's non-zero and zero outcomes with the producing-path constraints, enabling subsequent value-flow queries to recover implicit control-flow constraints.

$$\mathrm{CD}(b,o) = \{x \mid \mathrm{succ}_o(b) \rightsquigarrow x \text{ before reaching } \mathrm{ipdom}(b)\} \quad \mathrm{end}(P) = v$$

$$\mathcal{B}_{\mathrm{path}}(P) = \bigwedge_{\substack{\langle b,o\rangle \in P \\ v \in \mathrm{CD}(b,o), o \in \{1,0\}}} \mathcal{B}_o(b)$$

$$\mathcal{B}_q(v) = \bigvee_{\substack{P_i:\mathrm{entry} \rightsquigarrow b_i \\ b_i \text{ contributes } x_i \text{ to } v}} \left(\mathcal{B}_{\mathrm{path}}(P_i) \wedge \mathcal{B}_q(x_i)\right), \qquad q \in \{1,0\}$$

$$\mathcal{B}_q(c) = \begin{cases} \top, & \chi(c) = q, \\ \bot, & \chi(c) \neq q. \end{cases} \qquad \chi(c) = \begin{cases} 1, & c \neq 0, \\ 0, & c = 0. \end{cases} \qquad c \text{ is constant.}$$



Deriving implicit control flow constraints from value flow

Compose control-dependent branch impacts into path constraints during CFG traversal.

Merge path states that reach the same basic block with identical control-dependent branch-decision histories to improve efficiency.

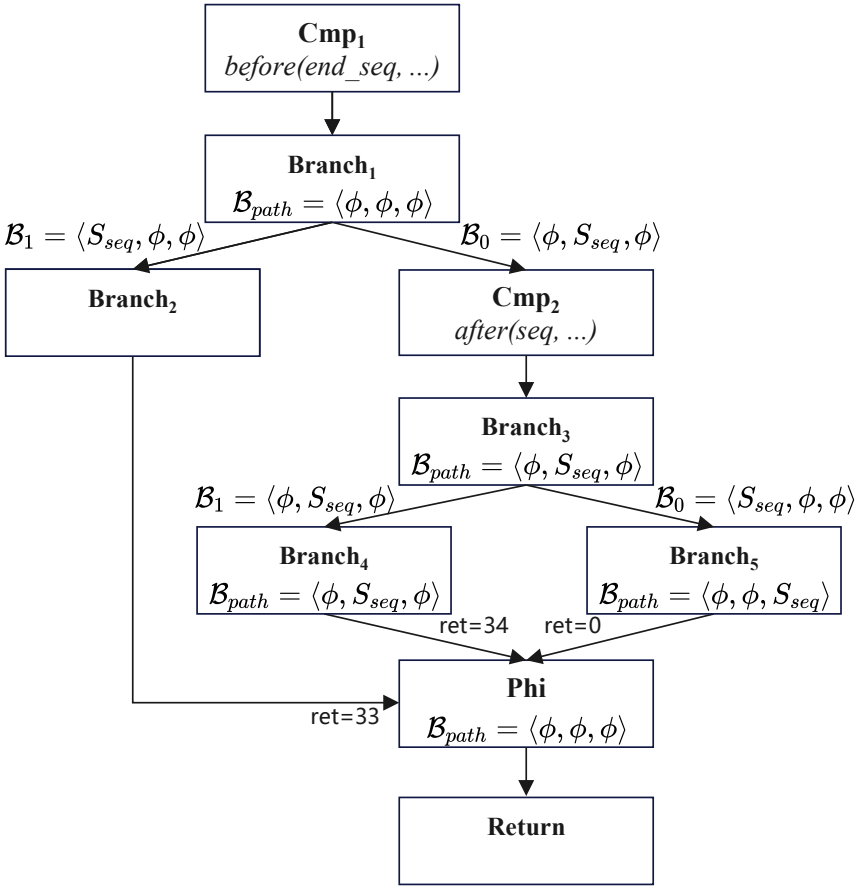
Meanwhile, associate each value's non-zero and zero outcomes with the producing-path constraints, enabling subsequent value-flow queries to recover implicit control-flow constraints.

$$\mathrm{CD}(b,o) = \{x \mid \mathrm{succ}_o(b) \rightsquigarrow x \text{ before reaching } \mathrm{ipdom}(b)\} \quad \mathrm{end}(P) = v$$

$$\mathcal{B}_{\mathrm{path}}(P) = \bigwedge_{\substack{\langle b,o\rangle \in P \\ v\in \mathrm{CD}(b,o), o\in\{1,0\}}} \mathcal{B}_o(b)$$

$$\mathcal{B}_q(v) = \bigvee_{\substack{P_i:\mathrm{entry}\rightsquigarrow b_i \\ b_i \text{ contributes } x_i \text{ to } v}} \left(\mathcal{B}_{\mathrm{path}}(P_i) \wedge \mathcal{B}_q(x_i)\right), \qquad q \in \{1,0\}$$

$$\mathcal{B}_q(c) = \begin{cases} \top, & \chi(c) = q, \\ \bot, & \chi(c) \neq q. \end{cases} \qquad \chi(c) = \begin{cases} 1, & c \neq 0, \\ 0, & c = 0. \end{cases} \qquad c \text{ is constant.}$$



Deriving implicit control flow constraints from value flow

Compose control-dependent branch impacts into path constraints during CFG traversal.

Merge path states that reach the same basic block with identical control-dependent branch-decision histories to improve efficiency.

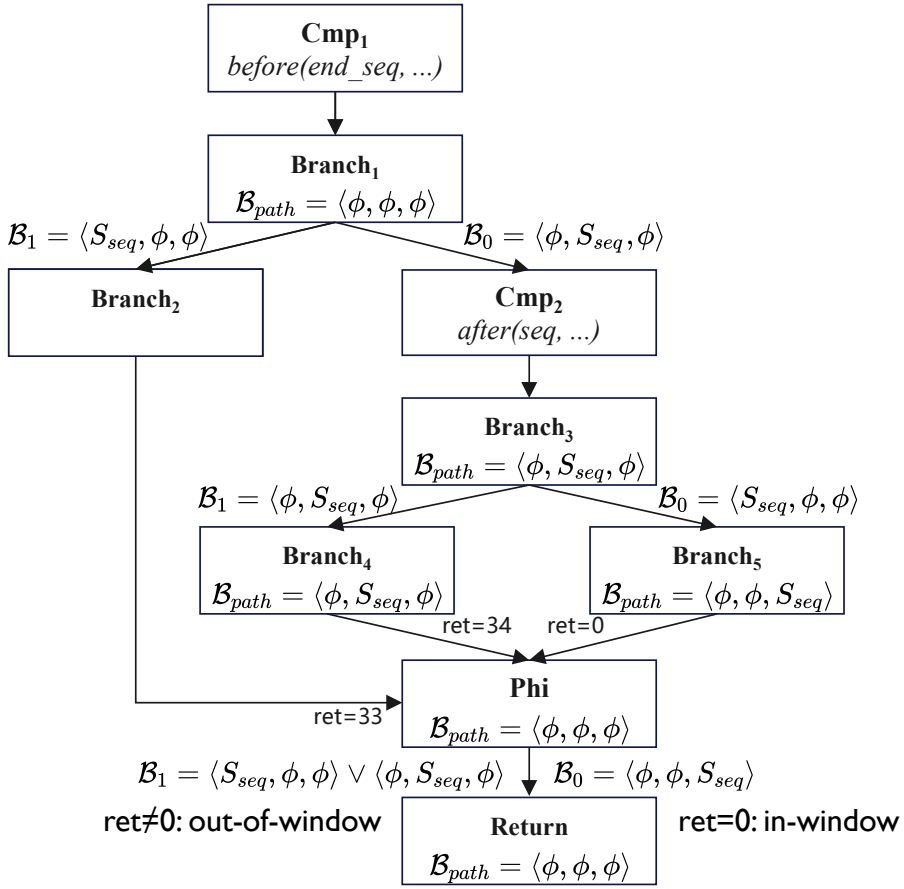
Meanwhile, associate each value's non-zero and zero outcomes with the producing-path constraints, enabling subsequent value-flow queries to recover implicit control-flow constraints.

$$\mathrm{CD}(b,o) = \{x \mid \mathrm{succ}_o(b) \rightsquigarrow x \text{ before reaching } \mathrm{ipdom}(b)\} \quad \mathrm{end}(P) = v$$

$$\mathcal{B}_{\mathrm{path}}(P) = \bigwedge_{\substack{\langle b,o\rangle \in P \\ v\in \mathrm{CD}(b,o), o\in\{1,0\}}} \mathcal{B}_o(b)$$

$$\mathcal{B}_q(v) = \bigvee_{\substack{P_i:\mathrm{entry}\rightsquigarrow b_i \\ b_i \text{ contributes } x_i \text{ to } v}} \left(\mathcal{B}_{\mathrm{path}}(P_i) \wedge \mathcal{B}_q(x_i)\right), \qquad q \in \{1,0\}$$

$$\mathcal{B}_q(c) = \begin{cases} \top, & \chi(c) = q, \\ \bot, & \chi(c) \neq q. \end{cases} \qquad \chi(c) = \begin{cases} 1, & c \neq 0, \\ 0, & c = 0. \end{cases} \qquad c \text{ is constant.}$$



Use function summaries to cache local path constraints

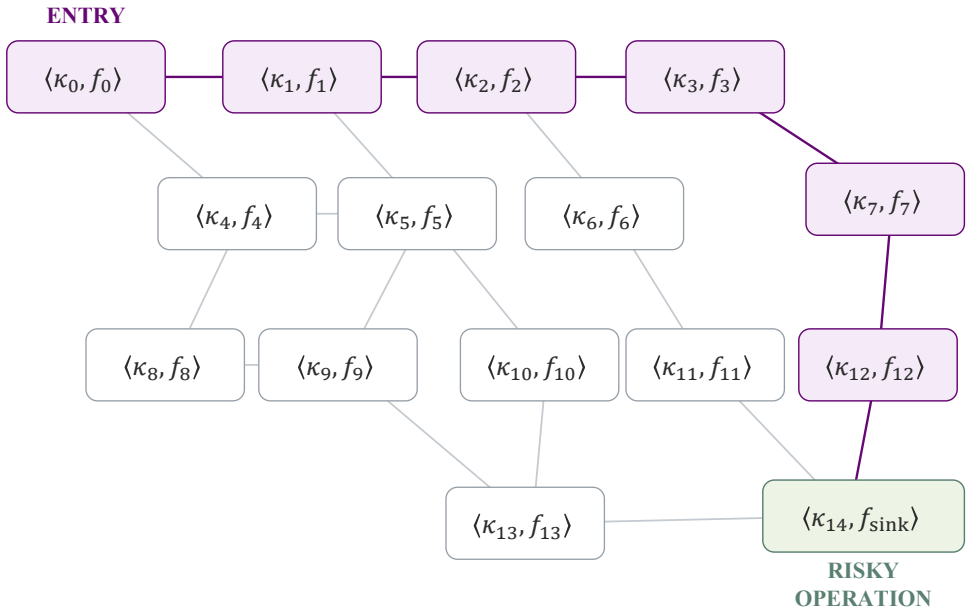
DEFINITION 3 — FUNCTION SUMMARY

$$S(\kappa, f, bb_d) = \left\{ \left(P_i, \mathcal{B}_{\text{path}}(P_i) \right) \mid \text{start}(P_i) = \text{entry}(f) \wedge \text{end}(P_i) = bb_d \right\}$$

A function summary for a specific basic block bb_d under context κ records deduplicated paths ending at it and their path constraints, designed to avoid re-expanding local CFGs across caller contexts.

Build Summary Graph based on function summaries

$$\frac{cs \in bb_{cs}, \quad \text{callee}(cs) = g, \quad \kappa' = \text{UpdateCtx}(\kappa, cs), \quad \left(P_i, \mathcal{B}_{\text{path}}(P_i) \right) \in S(\kappa, f, bb_{cs})}{\langle \kappa, f \rangle \xrightarrow[cs]{P_i,\ \mathcal{B}_{\text{path}}(P_i)} \langle \kappa', g \rangle \in E_{\text{SG}}}$$



Compose the interprocedural path constraint

$$G_{\text{SG}} = (V_{\text{SG}}, E_{\text{SG}}), \qquad \Pi = (e_1, \ldots, e_m) \in \text{Paths}_{\text{entry} \to \text{sink}}(G_{\text{SG}}), \qquad e_i \in E_{\text{SG}}$$

$$w(e_i) = \mathcal{B}_{\text{path}}(P_i), \qquad \mathcal{B}_{\text{inter}}(\Pi) = \bigwedge_{i=1}^{m} w(e_i) = \bigwedge_{i=1}^{m} \mathcal{B}_{\text{path}}(P_i)$$

Three filters make whole-stack weak-path search practical

I. Header Activation

Begin tracking a protocol header only after the interprocedural path enters its handler.

do not track tcp header → `tcp_v4_rcv()` → start tracking tcp header

II. Constraint Deduplication

Retain only outcomes that add new directional constraint information in the history used for path deduplication.

$\mathcal{B}_{\text{path}}(P) \wedge \mathcal{B}_o(b) \neq \mathcal{B}_{\text{path}}(P) \Rightarrow$ retain $\langle b, o \rangle$ in the deduplication history,
$\mathcal{B}_{\text{path}}(P) \wedge \mathcal{B}_o(b) = \mathcal{B}_{\text{path}}(P) \Rightarrow$ omit $\langle b, o \rangle$ from the deduplication history.

III. Strength Filtering

Reports a path as weakly verified when one feasible alternative strongly constrains at most τ input bytes and contains no highly secure per-flow field (critical field).

$$Weak(\Pi) \Leftrightarrow \exists T \in B_{\text{inter}}(\Pi): |T.E| \leq \tau \wedge T.E \cap Critical = \emptyset$$

$$Critical = \{TCP\ seq/ack, DCCP\ seq/ack, SCTP\ vtag, \ldots\}$$

Evaluation Setup

Evaluated on our local server with an Intel Xeon Gold 5418Y CPU and 500 GB RAM.

Experimented on both IPv4 and IPv6 stacks in Linux 5.15, Linux 6.8 and FreeBSD 14.1, supporting 6 protocols and 2 types of sinks.

The strong-byte filtering threshold τ is set to 8, calibrated to the case in [Feng 2023][1].

BUILDS AND ANALYSIS COVERAGE

Build	Analyzed LoC	TPG nodes
Linux 5.15 · IPv4	107,536	17,544,618
Linux 5.15 · IPv6	135,033	20,318,792
Linux 6.8 · IPv4	114,602	14,171,801
Linux 6.8 · IPv6	141,514	16,270,503
FreeBSD 14.1 · IPv4	94,149	18,881,472
FreeBSD 14.1 · IPv6	96,103	32,816,913

ANALYSIS BOUNDARY

Kernel / version	IP stack	Analyzed protocols	Routing-cache update sinks	Packet delivery / injection sinks
Linux 5.15	**IPv4**	ICMPv4 · TCP · UDP SCTP · DCCP	update_or_create_fnhe	tcp_queue_rcv · udp_queue_rcv_skb sctp_eat_data · dccp_enqueue_skb
Linux 5.15	**IPv6**	ICMPv6 · TCP · UDP SCTP · DCCP	rt6_insert_exception	tcp_queue_rcv · udpv6_queue_rcv_skb sctp_eat_data · dccp_enqueue_skb
Linux 6.8	**IPv4**	ICMPv4 · TCP · UDP SCTP · DCCP	update_or_create_fnhe	tcp_queue_rcv · udp_queue_rcv_skb sctp_eat_data · dccp_enqueue_skb
Linux 6.8	**IPv6**	ICMPv6 · TCP · UDP SCTP · DCCP	rt6_insert_exception	tcp_queue_rcv · udpv6_queue_rcv_skb sctp_eat_data · dccp_enqueue_skb
FreeBSD 14.1	**IPv4**	ICMPv4 · TCP · UDP SCTP	tcp_hc_updatemtu · rib_add_redirect	sbappendstream_locked · udp_append sctp_process_data
FreeBSD 14.1	**IPv6**	ICMPv6 · TCP · UDP SCTP	tcp_hc_updatemtu · rib_add_redirect	sbappendstream_locked · udp6_append sctp_process_data

[1] Feng et al. Man-in-the-middle attacks without rogue ap: Whenwpas meet icmp redirects. In 2023 IEEE Symposiumon Security and Privacy (SP).

Evaluation — Four questions guide the evidence

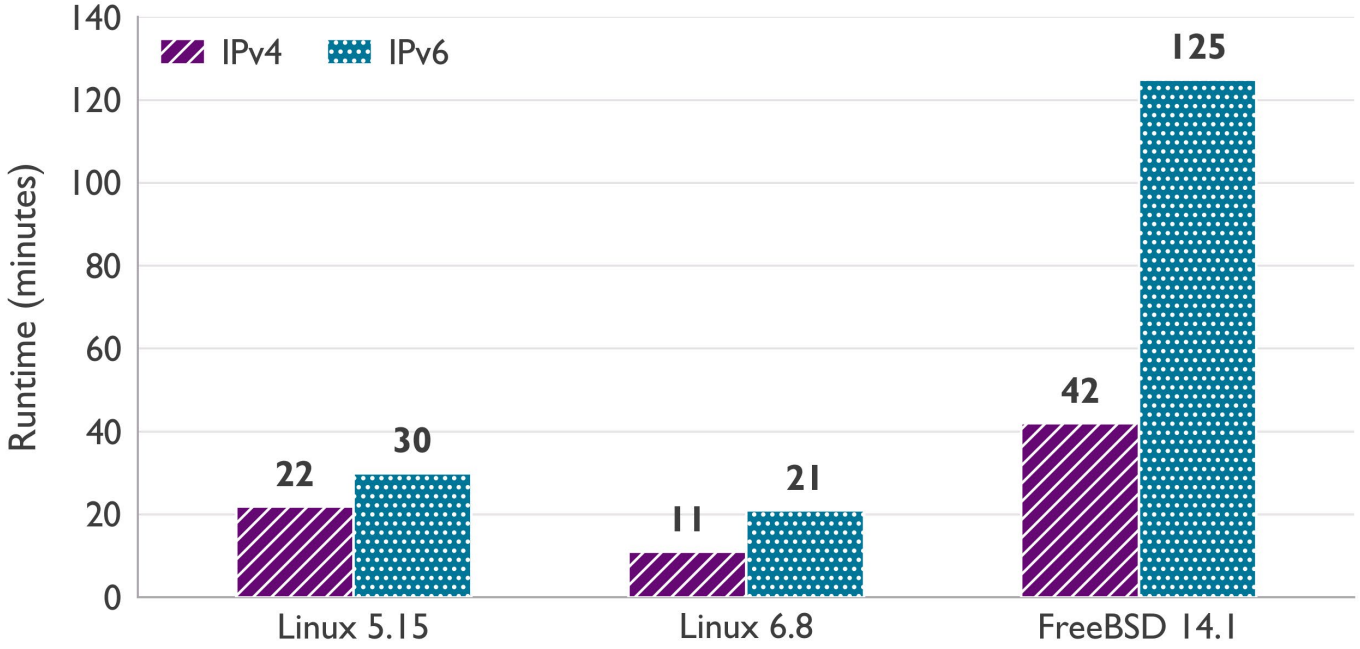
RQ1 · EFFICIENCY	How efficiently does **WAVED** analyze full protocol implementations?
RQ2 · EFFICACY	Can **WAVED** uncover practical weak-verification vulnerabilities?
RQ3 · PRECISION	How precise are **WAVED**'s reports?
RQ4 · REDUCTION	How much does **WAVED** reduce candidate paths?

RQ1 — How efficiently does **WAVED** analyze full protocol implementations?

11–125

minutes

Large-scale protocol stack analysis completes in about two hours on the evaluated hardware, demonstrating practical scalability for full protocol implementations.



RQ2 — Can **WAVED** uncover practical weak-verification vulnerabilities?

19 vulnerabilities · 14 previously unknown

■ = previously reported · no fill = new

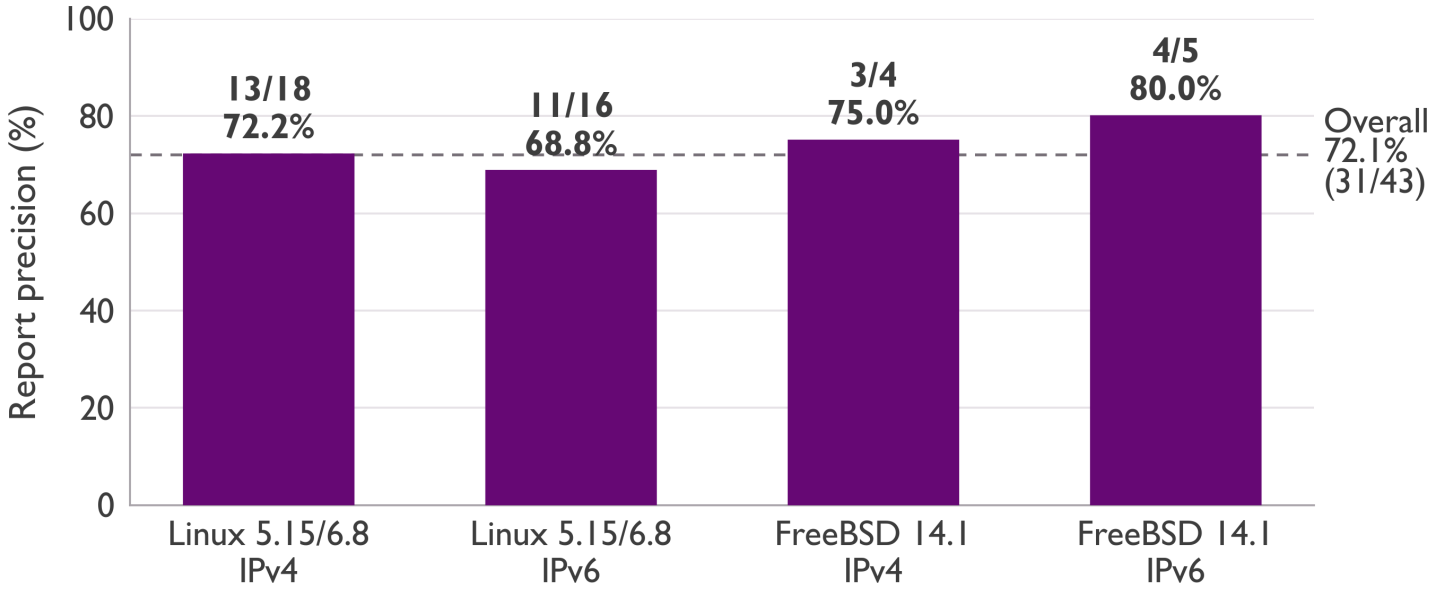
#	Protocol	Packet Form / Type	Attack Constraints	Attack Effect	Affected System
1	ICMPv4	Redirect [Echo Reply]	No extra constraint	gw cache poisoning	Linux 5.15 & Linux 6.8
2	ICMPv4	Redirect [UDP]	UDP sport match (UDP dport match when connected)	gw cache poisoning	Linux 5.15 & Linux 6.8
3	ICMPv4	Redirect [IP]	RAW socket of inner protocol	gw cache poisoning	Linux 5.15 & Linux 6.8
4	ICMPv4	Redirect [Echo Request]	Ping ident match	gw cache poisoning	Linux 5.15 & Linux 6.8
5	ICMPv4	Fragment Needed [Echo Reply]	No extra constraint	pmtu cache poisoning	Linux 5.15 & Linux 6.8
6	ICMPv4	Fragment Needed [UDP]	UDP sport match (UDP dport match when connected)	pmtu cache poisoning	Linux 5.15 & Linux 6.8
7	ICMPv4	Fragment Needed [IP]	RAW socket of inner protocol	pmtu cache poisoning	Linux 5.15 & Linux 6.8
8	ICMPv4	Fragment Needed [Echo Request]	Ping ident match	pmtu cache poisoning	Linux 5.15 & Linux 6.8
9	ICMPv4	Redirect [IP]	No extra constraint	gw cache poisoning	FreeBSD 14.1
10	ICMPv6	NDISC Redirect [ICMPv6]	No extra constraint	gw cache poisoning	Linux 5.15 & Linux 6.8
11	ICMPv6	NDISC Redirect [UDPv6]	UDP sport match (UDP dport match when connected)	gw cache poisoning	Linux 5.15 & Linux 6.8
12	ICMPv6	NDISC Redirect [IPv6]	RAWv6 socket of inner protocol RECVERR option set or connected	gw cache poisoning	Linux 5.15 & Linux 6.8
13	ICMPv6	Packet Too Big [ICMPv6]	No extra constraint	pmtu cache poisoning	Linux 5.15 & Linux 6.8
14	ICMPv6	Packet Too Big [UDPv6]	UDP sport match (UDP dport match when connected)	pmtu cache poisoning	Linux 5.15 & Linux 6.8
15	ICMPv6	Packet Too Big [IPv6]	RAWv6 socket of inner protocol RECVERR option set or connected	pmtu cache poisoning	Linux 5.15 & Linux 6.8
16	ICMPv6	NDISC Redirect [IPv6]	No extra constraint	gw cache poisoning	FreeBSD 14.1
17	ICMPv6	Packet Too Big [IPv6]	No extra constraint	pmtu cache poisoning	FreeBSD 14.1
18	DCCPv4	RST / CLOSEREQ / CLOSE	sock in DCCP_REQUESTING DCCP port match	Denial of Service	Linux 5.15 & Linux 6.8
19	DCCPv6	RST / CLOSEREQ / CLOSE	sock in DCCP_REQUESTING DCCP port match	Denial of Service	Linux 5.15 & Linux 6.8

RQ3 — How precise are **WAVED**'s reports?

72.1%

report-level precision

Reports are path outputs. Semantic consolidation produces the 19 distinct vulnerabilities shown in RQ2.



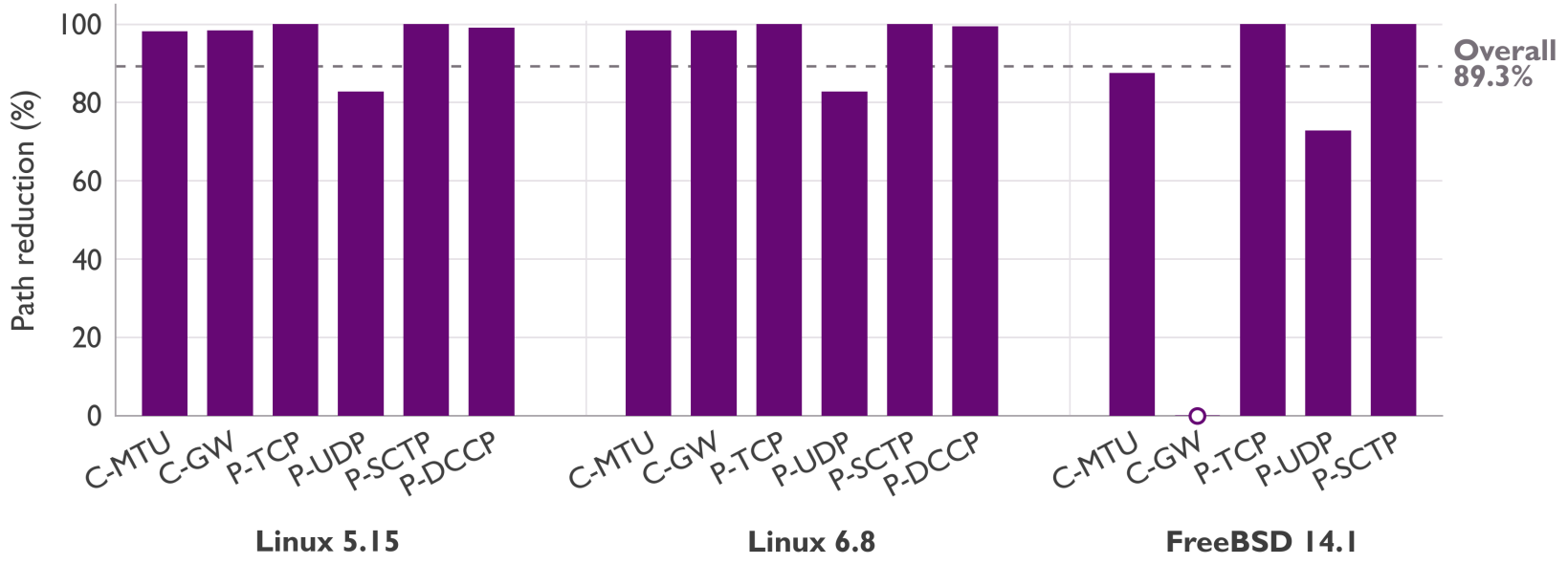
RQ4 — How much does **WAVED** reduce candidate paths?

89.3% fewer candidate paths on average than traditional implicit taint analysis.

89.3%

average path reduction

Fewer candidate paths reduce manual inspection effort.



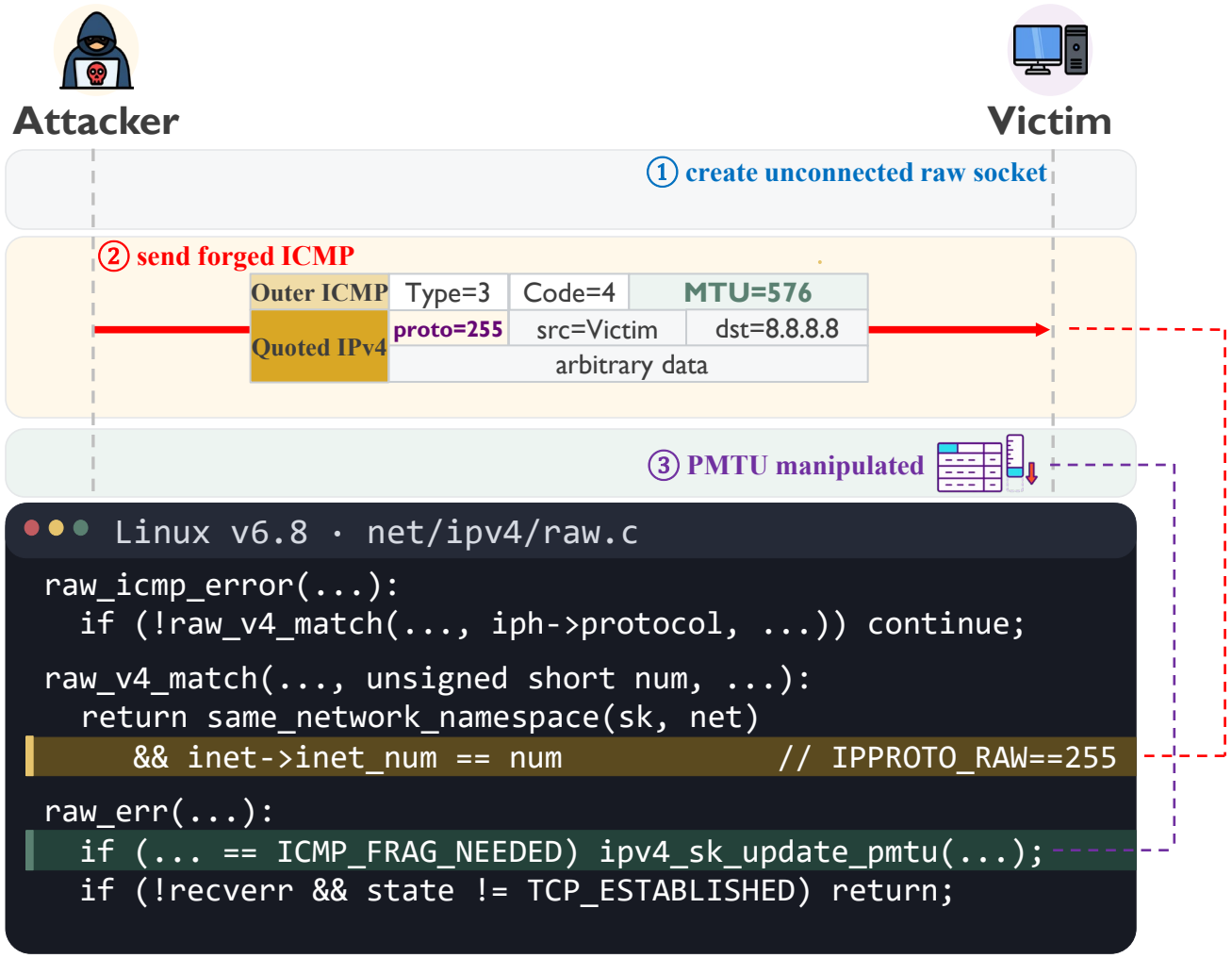
Case Study: Forged ICMP poisons PMTU via an unconnected raw socket

① An unconnected raw socket is active on the victim host (e.g. using Nmap)

② The attacker forges an ICMPv4 Fragment Needed (Type=3, Code=4) with a quoted IPv4 packet (proto=255)

③ Protocol 255 has no defined upper-layer protocol, such packets can only be malicious. But a forged ICMP quoting it can manipulate the PMTU cache without extra validation

Assigned CVE-2026-46266



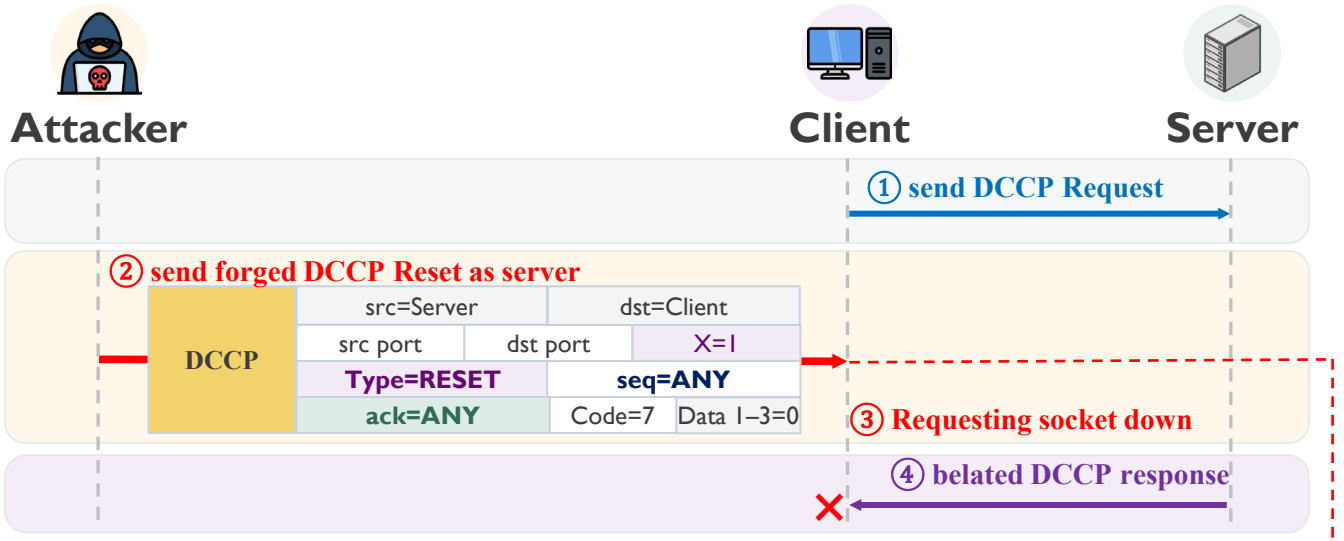
Case Study: Forged DCCP Reset aborts REQUEST without ACK validation

① A DCCP client sends a DCCP Request to start a connection to the server

② The attacker spoofs a DCCP Reset with matching ports and arbitrary 48-bit SEQ/ACK to the client

③ No additional validation is performed except for the ports (an out-of-window ACK can pass the check when in DCCP_REQUESTING), and the connection is reset, violating RFC 4340

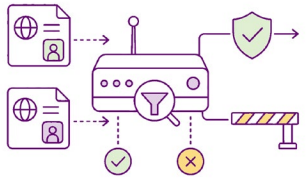
④ The belated DCCP response arrives, but is dropped since the requesting socket is already closed



Linux v6.8 · net/dccp/input.c

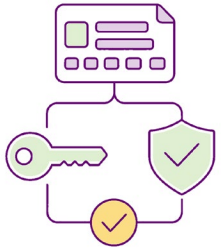
```
dccp_rcv_state_process(...)
  if (sk->sk_state != DCCP_REQUESTING &&
      dccp_check_seqno(sk, skb)) goto discard;
  ...
  if (dh->dccph_type == DCCP_PKT_RESET)
    dccp_rcv_reset(sk, skb); return 0;
```

Mitigation — Layer defenses before shared kernel state changes



Filter invalid source addresses

Routers should drop packets with invalid source addresses.



Validate protocol evidence

Require upper-layer secrets before executing risky operations.



Avoid unsafe state mutation

For stateless protocols, notify applications instead of modifying shared kernel state unreliably.

Source filtering is the strongest defense; otherwise validate control evidence before any shared-state mutation.

Summary

• We introduce the first systematic study of weak verification in TCP/IP stacks and propose our open-source tool, **WAVED**, to automatically discover weak-verification paths in a scalable way.

• Compared with previous studies, **WAVED** overcomes three specific limitations by extending the SVF framework to support more precise pointer analysis and byte-level data-flow analysis, direction-sensitive constraint measurement, and scalable path search.

• Across IPv4/IPv6 in Linux 5.15, Linux 6.8, and FreeBSD 14.1, **WAVED** found 19 vulnerabilities, including 14 previously unknown. Evaluations show that **WAVED** effectively reduces the number of candidate paths requiring manual inspection when discovering such vulnerabilities. We responsibly disclosed all our findings to the kernel maintainers and received acknowledgments.

Thank you!

Repository: https://github.com/Internet-Architecture-and-Security/WAVED

Email: zhaoyz24@mails.tsinghua.edu.cn