Static Detection of Packet Injection Vulnerabilities – A Case for Identifying Attacker-controlled Implicit Information Leaks

Qi Alfred Chen, Zhiyun Qian†, Yunhan Jack Jia, Yuru Shao, Z. Morley Mao
University of Michigan, †University of California, Riverside
alfchen@umich.edu, †zhiyunq@cs.ucr.edu, {jackjia, yurushao, zmao}@umich.edu

ABSTRACT

Off-path packet injection attacks are still serious threats to the Internet and network security. In recent years, a number of studies have discovered new variations of packet injection attacks, targeting critical protocols such as TCP. We argue that such recurring problems need a systematic solution. In this paper, we design and implement PacketGuardian, a precise static taint analysis tool that comprehensively checks the packet handling logic of various network protocol implementations. The analysis operates in two steps. First, it identifies the critical paths and constraints that lead to accepting an incoming packet. If paths with weak constraints exist, a vulnerability may be revealed immediately. Otherwise, based on "secret" protocol states in the constraints, a subsequent analysis is performed to check whether such states can be leaked to an attacker.

In the second step, observing that all previously reported leaks are through implicit flows, our tool supports implicit flow tainting, which is a commonly excluded feature due to high volumes of false alarms caused by it. To address this challenge, we propose the concept of attacker-controlled implicit information leaks, and prioritize our tool to detect them, which effectively reduces false alarms without compromising tool effectiveness. We use PacketGuardian on 6 popular protocol implementations of TCP, SCTP, DCCP, and RTP, and uncover new vulnerabilities in Linux kernel TCP as well as 2 out of 3 RTP implementations. We validate these vulnerabilities and confirm that they are indeed highly exploitable.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*; C.2.5 [**Computer-Communication Networks**]: Local and Wide-Area Networks—*Internet (e.g., TCP/IP)*

General Terms

Security, Program Analysis

Keywords

Network protocol security, Implicit information leakage, Static analysis, Side channel detection

CCS'15, October 12–16, 2015, Denver, CO, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813643.

1. INTRODUCTION

The encryption coverage on today's Internet is unfortunately still rather poor: only 30% [46]. Thus, off-path packet injection attacks remain a serious threat to network security. Recently a number of such attacks and their variants have been reported including off-path TCP packet injection [18,19,37,38] and DNS cache poisoning attacks [34, 47]. These attacks jeopardize the integrity of network communication, and lead to serious damage where personal data from unsuspecting users can be leaked when visiting a web site. Despite application-layer encryption support (*e.g.,* SSL and TLS), network connections are still vulnerable. For instance, for HTTPS connections, the initial request sent by the browser may still be an unencrypted HTTP request, and the server subsequently redirects the client to the HTTPS site. As shown in a recent study [37], an off-path attacker can inject a legitimate response to the very first HTTP request. Furthermore, such packet injection attacks can result in DoS, *e.g.,* by injecting a reset (RST) packet with an inferred TCP sequence number.

To combat such threats, the network stacks typically implement stringent checks on various fields to verify if an incoming packet is valid. In fact, a number of RFCs like RFC 5961 [39] are dedicated to this purpose. However, two problems remain. First, the design of an RFC may not be formally verified to be secure. Second, even if the design is secure, the actual implementation may not always conform to the design. In fact, the implementation is generally much more complex and difficult to get right. For instance, it has been shown that TCP implementations on Linux and FreeBSD are significantly weaker than what the RFC recommends regarding the mitigation against off-path attacks [38]. This calls for a systematic approach to verify protocol implementations.

In this work, we fulfill this very need by developing an effective and scalable static program analysis tool, *PacketGuardian*, which can systematically evaluate the robustness (*i.e.,* the level of security strength) of a network protocol implementation against off-path packet injection attacks. To ensure effectiveness and accuracy, our tool uses a precise context-, flow-, and field-sensitive taint analysis with pointer analysis support. To handle the scalability challenge caused by such high sensitivity, we choose a data flow analysis of summary-based approach, which is known to be more scalable compared to other frameworks [43], and is demonstrated to scale to very large programs like the Linux kernel [52].

At a high level, the tool operates by performing analysis in two steps: (1) Find all paths leading to the program execution point of accepting an incoming packet. This helps identify the critical checks that a protocol implementation relies on to prevent packet injection, and may directly reveal a packet injection vulnerability if any check is weak. (2) Motivated by the observation that strong checks typically rely on certain hard-to-guess or "secret" commu-

nication protocol state, *e.g.,* TCP sequence numbers, or RTP source IDs, we perform a subsequent analysis to check whether such secret states can be leaked to an attacker through side channels.

In network protocol implementations, these "secret" protocol states are unlikely to be leaked directly through explicit flows, and all previously reported leakage has been through implicit flows [18, 37, 38]. Therefore, PacketGuardian supports implicit flow tainting, which is known to be of much less value compared to explicit flow tracking (implicit flow usually leaks at most 1 bit of information) and at the same time cause large numbers of false positives [28]. It is thus a commonly excluded feature in nearly all taint analysis tools [4, 17, 21, 25]. To address the false positive challenge without compromising tool effectiveness, we leverage a key insight that the previously-discovered practical leaks are all *attacker-controlled implicit information leaks*, meaning that an attacker can influence which bit to leak. By prioritizing this special type of leak, we effectively reduce the false positive number and make the tool more useful for finding practical vulnerabilities.

Our analysis requires access to source code, which is a realistic assumption for many key network protocols. The tool we have developed is fully functional and is able to analyze arbitrary portions of the Linux kernel source code. By applying our tool to the Linux kernel TCP, SCTP, DCCP, and variants of open source RTP protocol implementations, we are able to identify a set of new vulnerabilities not previously reported. For example, for the 3 RTP implementations, two can be compromised by injecting less than 51 packets. For the Linux kernel TCP implementation, our tool identifies 17 high-entropy protocol state leakage, with 11 of them successfully validated in a realistic test bed. This illustrates that the Linux kernel TCP stack is still vulnerable even after the recent patches for the previous known leakage [15, 39], indicating the complex nature of the problem.

The contributions of this paper are as follows:

- We formulate the problem to systematically analyze the security properties of network protocol implementations against off-path packet injection attacks, and develop an effective and scalable static program analysis tool to address it using a precise context-, flow-, and field-sensitive taint analysis with pointer analysis.
- To enable the detection of practical information leaks due to implicit flows while ensuring low false positives, we propose the concept of attacker-controlled implicit information leaks and prioritize our tool to detect them. To the best of our knowledge, we are the first to design a taint analysis tool for detecting attacker-controlled implicit information leaks.
- We implement and apply our tool on 6 real implementations for 4 network protocols. From the result, we are able to discover new and realistic vulnerabilities confirmed by proof-of-concept attacks for Linux kernel TCP and 2 out of 3 RTP implementations.

2. ATTACK THREAT MODEL

Fig. 1 depicts the threat model for the off-path packet injection attack considered in this paper. As shown, an existing communication channel (*e.g.,* a TCP connection, a UDP session, or RTP session) is established between Alice and Bob. The attacker's goal is to inject a packet into the channel targeting Bob, pretending to be a packet from Alice. The attack goal can be to inject payload, *e.g.,* to launch attack such as phishing, or to trigger the termination of the channel, resulting in denial-of-service (DoS). The attacker in this threat model is off-path, *i.e.,* much weaker and more realistic than a man-in-the-middle attacker. To ensure channel integrity, Alice and Bob usually share several secret protocol states, denoted as *s* in the figure, and include it in the packet. These states are unknown to the off-path attacker and should be hard to guess.

To incorporate recently-discovered packet injection vulnerabilities [18, 19, 37, 38], our threat model also *optionally* considers a collaborative attacker sharing the same system as Bob. This collaborative attacker can be an unprivileged malware program [37, 38], or a script in the browser [18, 19]. This collaborative attacker is tasked to provide feedback about any packet injection attempt of the off-path attacker, facilitating the inference of the secret protocol state for a successful injection.

3. ILLUSTRATIVE EXAMPLE

3.1 Packet Injection Attack for TCP

To illustrate how static analysis can help detect packet injection attacks for TCP, Fig. 2 shows a significantly simplified implementation example for handling an incoming TCP packet, which is the entry for an injection packet from an off-path attacker. This implementation is mostly based on Linux kernel 3.15, from which we only include the important logic, *i.e.,* sequence number and acknowledgment number checks. In this figure, `tcp_rcv_established()` is the main entry function, parameter `tp` is the socket status maintained by the system, and parameter `skb` is the data structure for the incoming packet. Function `accept_payload()` copies the packet data into the application layer, indicating the acceptance of the incoming packet for this TCP connection, *i.e.,* a successful injection.

To evaluate the robustness of this implementation against off-path packet injection, the key question is **what strong checks exist to prevent an off-path injected packet from reaching `accept_payload()`**. As we can see in `tcp_rcv_established()`, 3 checks on line 2, 3, and 4 exist. The check on line 2 requires the incoming packet to have either ACK or RST bit set, which is easy to bypass by an attacker. The checks on line 3 and 4 call into `tcp_validate_incoming()` and `tcp_ack()`, and can be passed only if the former returns true, and the later returns a non-negative value. In `tcp_validate_incoming()`, to return true, the `seq` field of the incoming packet needs to fall into the receive window [`tp->rcv_nxt`, `tp->rcv_nxt + tp->win1`], and the size of this window is usually between 2^{14} to 2^{20}. `tp->rcv_nxt` is a protocol state unknown to an off-path attacker, thus it takes up to 2^{18} guesses to pass the check. In addition, for `tcp_ack()` to return a non-negative value, `ack_seq` needs to fall into [`tp->snd_una - tp->win2`, `tp->snd_nxt`]. Like `rcv_nxt`, `snd_una` and `snd_nxt` are also protocol states unknown to the attacker, making this check also hard to pass. Combined with the check in `tcp_validate_incoming()`, it takes up to $2^{36} = 68,719,476,736$ guesses for a single packet to be accepted, making it practically unexploitable. Therefore, these are important checks to prevent off-path attackers. In this paper, we use the number of packets needed for one injection as the metric for evaluating *off-path packet injection robustness* of a protocol implementation, denoted by N_{pkt}.

We note that the robustness strongly depends on the implementation details. As shown in the bottom-left rectangle of Fig. 2, before Linux 3.7, the ACK bit check was much weaker. In this case, off-path attacker can simply set the ACK bit to 0 to avoid the checks in `tcp_ack()`, resulting in a large reduction in N_{pkt} from 2^{36} to 2^{18}. This turns out to be a missing implementation of a check required by the protocol specification [15]. Thus, *even for a well-designed protocol, the corresponding implementation of it may not be robust against off-path packet injection attacks*.

If strong checks do exist, which usually depend on secret protocol states unknown to the attacker, a further question is

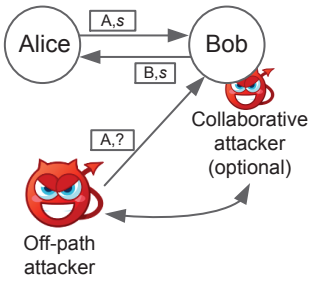

Figure 1: Packet injection attack threat model in this paper.

```
1: void tcp_rcv_established(tcp_sock* tp,
   sk_buff* skb) {
2:   If (!skb->ack && !skb->rst) return;
3:   If (!tcp_validate_incoming(tp, skb)) return;
4:   If (tcp_ack(tp, skb) < 0) return;
5:   accept_payload();
6: }
```

```
1: bool tcp_validate_incoming(tcp_sock* tp, sk_buff* skb) {
2:   If ((skb->seq >= tp->rcv_nxt ) && (skb->seq <= tp->rcv_nxt + tp->win1)) {
3:     tcp_send_dupack(tp, skb);
4:     return false;
5:   }
6:   return true;
7: }
```

```
1: void tcp_send_dupack(tcp_sock* tp, sk_buff* skb) {
2:   If (skb->seq < tp->rcv_nxt)
3:     tp->net_statistics[DelayedACKLost]++;
4: }
```

```
Version <= 3.7:
1: void tcp_rcv_established(tcp_sock* tp,
   sk_buff* skb) {
2:   If (!tcp_validate_incoming(tp, skb)) return;
3:   If (skb->ack && tcp_ack(tp, skb) < 0) return;
4:   accept_payload();
5: }
```

```
1: bool tcp_ack(tcp_sock* tp, sk_buff* skb) {
2:   If (skb->ack_seq < tp->snd_una) {
3:     if (skb->ack_seq < tp->snd_una - tp->win2) return -1;
4:     return 0;
5:   }
6:   If (skb->ack_seq > tp->snd_nxt) return -1;
7:   return 0;
8: }
```

Figure 2: An illustrative code example of a simplified implementation for handling an incoming TCP packet in Linux kernel 3.15.

whether with the help of a collaborative attacker, **these protocol states can be leaked**. This is of concern since previous work [18, 38] has shown that `rcv_nxt` and `snd_nxt` can have leakage through storage channels such as proc files. The threat demonstrated by Qian *et al.* [38] is especially realistic as `rcv_nxt` and `snd_nxt` can be inferred under only a second. The upper-right rectangle in Fig. 2 illustrates this reported leakage for `rcv_nxt`. Since it is very unlikely to pass the check on line 1 in `tcp_validate_incoming()`, the attack packet reaches `tcp_send_dupack()`, and if `seq` set by the attacker is smaller than `rcv_nxt`, it changes a counter *DelayedACKLost* in proc file, otherwise not. If we inspect this counter closely, each comparison leaks 1 bit of information, and thus at most 32 guesses/packets are needed to infer the exact value of `rcv_nxt`. Note that at the time of Qian *et al.* [38], the check in `tcp_ack()` is easy to bypass. In the current version, the check in `tcp_ack()` is strengthened as shown in the bottom-right rectangle in Fig. 2, and even if `rcv_nxt` is guessed, the code still does not have exploitable vulnerabilities for packet injection. However, from our automated vulnerability detection shown later in §7, we discover 14 new highly-exploitable leaks for `snd_nxt/snd_una` even after the fix. Thus, *even for well-implemented protocols with strong checks, the protocol states of these checks can still be leaked through attacker-accessible channels, rendering the checks ineffective*.

To systematically discover such vulnerabilities, we argue that automated analysis is required to ensure correctness and coverage, given that the implementations are rather complex — 64 different paths with more than 300 direct and 600 indirect checks are found before accepting an incoming packet in Linux kernel 3.15.8.

3.2 Attacker-controlled Implicit Information Leaks

In the above example, the leakage of protocol state `rcv_nxt` is one case of implicit information leaks as the secret is leaked through control dependency (predicates on line 2 in `tcp_send_dupack()`). Compared to classic implicit information leaks, this instance is quite special in that it involves attacker-controlled data in the predicates (`skb->seq` in the example), giving an attacker the ability to influence the control flow. We name this special type of leaks *attacker-controlled implicit information leaks*, a new concept proposed in this paper. As shown in the illustrative example, since attacker-controlled data is involved, an attacker can use different input to actively trigger leaks from the same predicate multiple times and thus extract the secret bit by bit, making it highly-exploitable in practice.

Table 1 shows a categorization of implicit information leaks to help illustrate the position of this new concept and this paper. Classic implicit information leaks is from a secret information related predicate (*e.g.,* `if (secret > 100)`) to an information sink (*e.g.,* a public value), which usually just leaks 1 bit of information (*e.g.,* whether `secret` is above 100 or not). Since the leakage volume is extremely low compared to explicit information leaks, and tracking it causes large numbers of false positives [5,28], detecting classic implicit information leaks is a commonly excluded feature in nearly all taint analysis tools [4, 17, 21, 25].

To enable the detection of severe information leaks from implicit flows without causing high volumes of false positives, Bao *et al.* propose to limit implicit flow tracking to a special type of control dependency called strict control dependency (SCD) [5]. SCD denotes the correlation between an equivalency predicate (*e.g.,* `if (secret == 100)`) and an information sink, thus when the information sink is changed, it directly reveals all bits of the secret, making it much more severe than the classic implicit information leaks. Cryptographic key extraction through cache side channels [22, 31, 54, 57] is one real-world exploit example of SCD-based leaks, which leverages the SCD in bitwise equivalence testing of the secret key in certain cryptographic system implementations such as RSA implementation of GnuPG [54, 57]. Another exploit example is side-channel leaks in web and Android applications [10, 11, 60], in which network traffic pattern is SCD on user choices in web or Android applications. As shown in Table 1, both examples are studied extensively on both attack and defense sides.

Attacker-controlled implicit information leaks is a newly-identified category of highly-exploitable implicit information leaks, and similar to Bao *et al.* [5], we propose to prioritize this special type of leaks in order to balance the vulnerability detection effectiveness and false positives. This concept is orthogonal to SCD in that attacker-controlled data is involved in the control dependency. The target of this paper is to identify exploitable cases of such leaks focusing on off-path packet injection attacks [18,37,38], and we are the first to design a taint analysis tool for detecting this type of leaks (detailed in §6).

4. PACKETGUARDIAN OVERVIEW

In this section, we first describe the analysis required for detecting packet injection vulnerabilities, and then present a design overview of PacketGuardian which supports this analysis.

4.1 Analysis Steps

Following the discussion in §3, we break the analysis into two steps: accept path analysis and protocol state leakage analysis.

Implicit information leak category	Exploitability	Example of exploits and related work		
		Exploit case	Attack	Detection/defense
Classic	Low	N/A	N/A	N/A
Strict control dependence (SCD) based [5]	High	Cryptographic key extraction	[22], [57],[54], [31]	[14], [58], [50], [40]
		Side-channel leaks in web/Android apps	[11], [10], [60]	[55], [9], [32]
Attacker-controlled	High	Off-path packet injection attack	[37], [38], [18]	***This paper***

Table 1: Categorization of implicit information leaks and position of this paper.

Step 1: Accept path analysis. For a packet injection, the goal is to pass all checks and reach the program point where the packet is accepted, *e.g.,* `accept_payload()` in Fig. 2. In this paper, we refer to these paths as *accept paths*. For a particular protocol implementation, the off-path packet injection robustness depends on the weakest accept path. Thus, the first analysis step is to find the weak accept paths in the implementation. The output needs to highlight the checks related to attacker-controlled information, *e.g.,* header fields, to help analyze the accept path strength.

Step 2: Protocol state leakage analysis. If all accept paths are all well-protected by "secret" protocol states unknown to the attacker, the implementation can still be vulnerable if these protocol states are vulnerable to information leakage as illustrated in §3. Thus, after accept path analysis, we follow up with an information leakage analysis for important protocol states.

The first step is to analyze the strength of the checks related to attacker input on the program path reaching a pre-defined analysis sink, which is similar to the traditional code injection analysis, and thus it can be modeled as a static taint analysis problem with attacker-controlled data as taint source like in previous work [25, 53, 59]. The second step is an information leakage problem and again can be solved by static taint analysis.

Note that symbolic execution is alternative choice, but since it tracks finer-grained information for each variable than taint analysis, it comes with much higher computation overhead, which is unlikely to be efficient and scalable enough in practice, especially in our case high analysis sensitivity are necessary (shown in §7.1). Thus, we choose taint analysis in the current design.

4.2 PacketGuardian Design

To support the analysis in §4.1, PacketGuardian has 2 major components: taint-based summarizer, and vulnerability analyzer, as shown in Fig. 3. In this section, we briefly introduce the design of each component, and details are provided in §5 and §6.

Pre-processing. To support taint analysis, the source code needs to be first pre-processed to the format required by a certain static analysis tool. We choose CIL [35] for our analysis, so for its input requirement, `.c` files are pre-processed to `.i` files in this step.

Taint-based summarizer. With pre-processed source code, given an entry function, taint-based summarizer performs a precise static taint analysis with flow, field, and context sensitivity with pointer analysis. In §7.1, we show that such analysis strength is required to discover real vulnerabilities with minimum false positives (FPs). Further, we employ implicit flow tracking (with separate taints from explicit flows), as the protocol logic checks commonly induce leakage through control dependence (see §3). Note that implicit tainting is known to generate a large number of FPs [28], and nearly all existing taint analysis tools choose to ignore implicit flows [4, 17, 21, 25]. We show that after prioritizing attacker-controlled implicit information leaks, PacketGuardian does not suffer from the excessive FP problem.

To achieve context sensitivity, our static taint analysis needs to be performed in an inter-procedural data flow analysis framework, with two major choices: IFDS/IDE framework [42, 44], and summary-based (or functional) approach [49]. IFDS/IDE framework performs analysis from function caller to callee, and in the worst case, the analysis complexity is proportional to the number of call graph edges. In contrast, summary-based approach first generates strongly-connected components (SCC) of the call graph and computes function summary from callee to caller. In this approach, each function only needs to be analyzed once and thus has lower complexity and significant performance gains [43]. Its disadvantage is that it needs storage for function summaries, and the callee-to-caller order makes taint path construction unnatural. To support high sensitivity and implicit flow tracking, our analysis faces a significant scalability challenge if applied to a large code base like the Linux kernel. Fortunately, as demonstrated in previous studies [52], summary-based approach can scale to very large programs.

Following these design choices, as shown in Fig. 3, all related source files are first crawled in a breath-first search framework starting from the entry function. After merging these files, function SCCs are computed and serve as input to the taint analysis engine. Taint analysis are then performed in the order of callee to caller, and output function summaries.

Vulnerability analyzer. In vulnerability analyzer, our tool uses the function summaries from the taint-based summarizer to construct paths for accept path analysis and protocol state leakage analysis in §4.1. Taking attacker-controlled data as taint source and packet accept functions as sink, *accept path constructor* constructs accept paths with the attacker-controlled data related predicates labeled. The output is further analyzed, with the result being either an obvious packet injection vulnerability, or a set of protocol states that the implementation relies on to prevent injection.

If the accept paths are well-protected by a set of protocol states, *leakage path constructor* performs the second step to find possible leakage of these important states. In this analysis, we also use the function summaries, but the taint sources and sinks become the protocol states and public side channels accessible to the attacker. These channels can be storage side channels [10, 24, 38, 60], public events like sending packets [18], timing, power, *etc.* Besides detecting leaks, we also construct the leakage paths to help tool users understand and analyze these leaks. In this step, we prioritize attacker-controlled implicit information leaks, as all previously reported highly-exploitable leaks are of this special type [18, 37, 38].

With the choice of summary-based approach, even though the taint sources and sinks are different in the two steps, our tool only needs to perform taint analysis, the most time-consuming part, once instead of multiple times for each source and sink pair. While identifying sources and sinks is a problem for taint analysis in general [41], PacketGuardian users can conveniently try different sinks in the analysis without re-running the taint analysis.

Manual effort in analysis. In our design, the manual effort mainly lies in identifying protocol states, and the amount of it depends on the number of output paths and predicates. As detailed in §6, our design mitigates this problem using path pruning and taint information annotations, which is shown to be effective in §7, *e.g.,* our pruning reduces 42.6% paths on average.

5. TAINT-BASED SUMMARIZER

In this section, we detail the two core designs of the taint-based summarizer, the taint analysis engine and function summary.

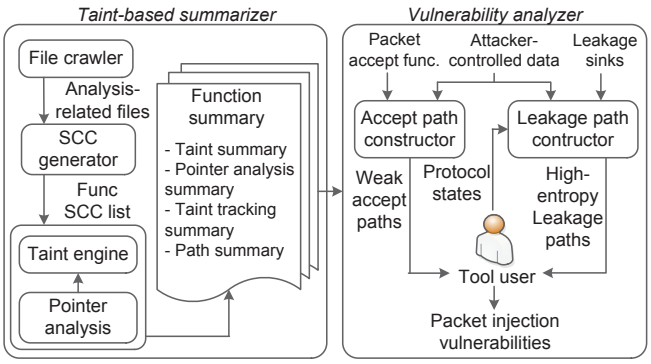

Figure 3: PacketGuardian design overview.

5.1 Taint Analysis Engine

In this section, we detail the design of taint environment, propagation logic, and how we support flow, context, and field sensitivity with pointer analysis.

Taint environment. To specify the tainting relationship, each program variable v is associated with a taint environment $\gamma : v \rightarrow T$, where T is a set of taint values $\{t_i | i = 1, ..., k\}$. Each taint value t_i is associated with a variable v_i, meaning that v is tainted by variable v_i. In our design, variables in γ include local, global, formal, and function return variables. Each v is specified by a tuple with its identification information such as variable name and type.

Taint label of explicit and implicit flows. As discussed in §4.2, it is a design requirement to include implicit flows, which is known to cause excessive FPs [28]. At the same time, the importance of explicit leaks is much higher than implicit leaks since the former directly leaks the entire data. Thus, to distinguish leaks of different importance and be able to support policies on limiting implicit flow tainting [26], we label each taint value with 2 boolean values d and c, for taint values coming from explicit flows ($d = true$) or implicit flows ($c = true$). This is a unique design in PacketGuardian and not supported in most existing taint analysis tool [4, 17, 21, 25].

Taint propagation. The tainting process is to propagate taint values by updating $\gamma(.)$ after processing each statement. Table 2 shows the taint propagation logic in the statement and expression format defined by CIL [35]. This table only has intra-procedure propagation logic, and inter-procedure logic will be covered later.

In the table, we introduce 3 new operations for taint label management, L_d, L_c, and $\cup^l$. L_d and L_c modify the labels of all taint values in a set with explicit flow and implicit flow label respectively, and $\cup^l$ is simply the set union operation but with label merging, for example if both sets have v but with different labels d_i, c_i and d_j, c_j, the merged taint value label is $(d_i||d_j)$ and $(c_i||c_j)$.

Flow-sensitive tainting with both explicit and implicit flows. Our taint propagation is performed in a data flow analysis framework, where each $stmt_i$ has a taint environment $\gamma_i(.)$, and after tainting according to the rules in Table 2, $\gamma_i(.)$ is updated and passed to the egress statements in CFG. Our data flow analysis is a may-taint analysis to tradeoff potentially higher FPs for lower FNs (we have other mechanisms to lower FPs later on). To increase the analysis efficiency, we use topology order to visit CFG nodes.

To support implicit flow tainting, we maintain a constraint path, CT, during the data flow analysis. CT describes the list of conditional branch statements such as `if` exp and `Switch` exp, which we call constraints (denoted by ct), that the current statement is control dependent on. Each ct is described by a tuple $\{exp, T_{exp}\}$, and adds a new ct after processing a conditional branch statements with exp. We compute the control dependence relationship with a postdominator analysis [49], and delete the ct from CT if the current statement is not control dependent on it. With this constraint list, we compute the implicit flow taint value set by merging T_{exp} of all ct in CT. As shown in Table 2, this implicit flow taint is added in taint propagation after applying $L_c(.)$.

Context sensitivity. To support context sensitivity, function call statements need to be correctly handled for inter-procedure taint propagation. According to our design, the taint modifications after calling a callee function can also be described in a function taint environment $\gamma^f(.)$, by merging the return statement taint environments in the callee function using $\cup^l$ operation.

Before being applied, $\gamma^f(.)$ needs to be transformed to the caller function context since $\gamma^f(.)$ is computed in the callee context. This transformation is done in an instantiate function $Inst : v_{callee} \rightarrow v_{caller}$, which replaces the formal parameter variables in callee function with the caller actual parameter variables in the call site of caller function. $Inst(.)$ also handles the side effect in the process for the callee function variables, *i.e.,* caused by changing the values of de-referenced pointer formal or global parameter variables, using a context-sensitive pointer analysis explained later.

Field sensitivity. As shown in the example in §3, the header fields related to protocol states in a network protocol are usually implemented as a few fields of a composite type variable. Thus, it will cause large numbers of FPs if we don't distinguish same variable with different fields and taint the whole variable like in some previous tools [2, 17]. We support field sensitivity with the standard technique of expanding each variable with an offset element in the variable tuple. After adding this feature, both the intra- and inter-procedure taint propagation logic need to be updated accordingly.

Adding offset element in variable tuple can also cause $\gamma^f(.)$ to keep increasing with same variable having different offset due to recursive fields (*e.g.,* `next` in linked list data structure) in a loop. To solve this problem, we add an iteration limit of loops, which is a common practice in field-sensitive data flow analysis.

Taint with pointer analysis. As shown in §3, network protocol implementations use pointer extensively, and in our example, the leakage sink is changed with de-referencing a pointer, making pointer analysis a must. In our design, we choose pointer analysis to support referencing and de-referencing pointers when needed during taint propagation. To better work with our taint analysis, our pointer analysis is also flow-, field-, and context-sensitive based on the traditional flow-sensitive pointer analysis framework [23].

With this feature, our analysis has another environment, *pointer environment*, $Ptr : v \rightarrow \{v_i | i = 1, ..., k\}$, meaning that v points to a set of variables $\{v_i | i = 1, ..., k\}$. Like taint environment, we associate each statement $stmt_i$ with a pointer environment for flow-sensitive analysis. In inter-procedure case, the pointer relationship in a callee function is summarized to a a function pointer environment Ptr^{callee}, and $Inst(.)$ is also needed to transform the variables to caller function context accordingly.

Note that parameter aliasing is a classic problem in summarizing points-to relationship, which is typically solved by partial transfer functions (PTF) [51]. In network protocol implementations, pointer parameters typically are used for semantically different purposes, *e.g.,* `tp` for socket status and `skb` for the incoming packet in the illustrative example (§3), thus we assume no parameter aliasing in the current implementation. This may introduce inaccuracies, and we plan to implement PTF for improvement in future work.

5.2 Function Summary

After taint analysis, the function summary are generated with 4 parts : taint summary, pointer summary, taint tracking summary, and path summary.

Statement/expression	Taint operation
$Const, Sizeof(typ/str)$	$T_{exp} = \varnothing$
v	$T_{exp} = L_d(\gamma(v))$
$Sizeof(exp_1)$	$T_{exp} = L_d(T_{exp_1})$
$Cast(exp_1)$	$T_{exp} = L_d(T_{exp_1})$
$unop(exp_1)$	$T_{exp} = L_d(T_{exp_1})$
$biop(exp_1, exp_2)$	$T_{exp} = L_d(T_{exp_1}) \cup^l L_d(T_{exp_2})$
$exp : exp_1?exp_2 : exp_3$	$T_{exp} = L_c(T_{exp_1}) \cup^l L_d(T_{exp_2}) \cup^l L_d(T_{exp_3})$
$v = exp$	$\gamma(v) = L_d(T_{exp}) \cup^l L_c(\cup^l\{T_{ct_k} \vert ct_k \in CT\})$
$Asm(\{exp_{in_i} \vert i\},$ $\{v_{out_j} \vert j\})$	$\gamma(v_{out_j}) = L_d(\cup^l\{T_{exp_{in_i}} \vert i)$ $\cup^l L_c(\cup^l\{T_{ct_k} \vert ct_k \in CT\})$

Table 2: Taint value calculation and propagation logic for intra-procedure propagation. CT includes the constraints that the current statement is control dependent on.

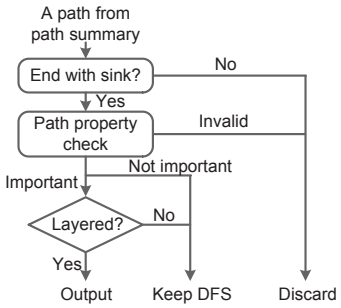

Figure 4: Path analysis process in DFS path construction and analysis framework.

Taint and pointer summaries. Taint and pointer summaries are the function taint environment $\gamma^f(.)$ and function pointer environment $Ptr^f(.)$ respectively (detailed in §5.1). After generated, they are fed back into the taint analysis engine for subsequent analysis to support inter-procedure taint and pointer analysis.

Taint tracking summary. As mentioned in §4.2, since we choose summary-based approach over IFDS/IDE for scalability, tracking taint propagation becomes unnatural. However, we do need this tracking since it benefits our vulnerability analysis by making the taint result explainable. Thus, we design taint tracking summary to fulfill this goal in PacketGuardian. Note that this summary has another important benefit for our analysis as it can help us locate the indirect constraints of implicit flow taint to obtain a complete accept path and leakage path (detailed later in §6).

Like function taint and pointer environments, this summary is specified by a tracking environment $Track : \langle v, t\rangle \rightarrow TR$, where $t = \gamma^f(v)$ and TR is the set of track values. Each track value describes one source for a taint value, which can come from intra-procedure explicit flow, intra-procedure implicit flow, or inter-procedure explicit or implicit flow from a callee function. Since explicit flow is relatively easy to understand, to lower the tracking overhead we only record the source file line numbers of the program point passing the taint. For implicit flows, we create a track value for each ct in CT to make it precise.

For inter-procedure taint tracking, we don't let the track value propagate from callee to caller function like in taint and pointer summaries. Otherwise, the track value set will increase accumulatively at each time of inter-procedure propagation, making the analysis hard to scale. More importantly, in that case each taint tracking summary will have complete taint history for each variable and taint value pair, which is unnecessary since only a few important variables need tracking. Thus, in our design, during function call we only store a "function pointer" in the TR, and delay the actual inter-procedure tracking computation till the vulnerability analysis phase when needed. This "function pointer" is designed to have complete context information to load the callee taint tracking summary and reconstruct the inter-procedure tainting path later.

Path summary. To meet the goal of outputting the accept and leakage paths for explaining the packet injection vulnerability, during the taint analysis we also summarize the important paths. Like taint tracking summary, recording the inter-procedure program paths is not necessary, and we only record the intra-procedure program paths, and keep a "function pointer".

To satisfy the analysis requirements, the path we record has 2 parts, a constraint path and a path end point. The constraint path is the same as CT mentioned earlier, and here the list of ct is those ones that the path end point is control dependent on. To help explain the path and also enable further tracking of the expression taint, we expand $ct = \{exp, T_{exp}\}$ with 3 elements: variable taint value set $\{\langle v_i, t_i, Track(\langle v_i, t_i\rangle)\rangle | i = 1...k\}$, branch br, and line number, where v_i is a variable used in exp. Variable taint value set gives fine-grained information about the taint values and track values for each variables used in ct, which helps the path pruning and prioritizing detailed later in §6. Branch br records whether this path takes the true branch of ct or the false branch of it.

The path end point can be in two forms: a function, or a sink-related statement. The path end point of a function is designed to serve for the role of "function pointer" mentioned earlier, and it can also serve for the vulnerability analysis with a function sink, *e.g.,* `accept_payload()` in Fig. 2. The path end point of a sink-related statement is designed to mainly serve for protocol state leakage analysis when this statement is related to a channel accessible to an off-path attacker. For example, this statement can be modifying a public value in storage channels [10, 37], or related to a special instruction in data timing channels (*e.g.,* SSE instructions discussed by Andrysco et. al. [3]), *etc.* In our current implementation, we focus on storage channels and record the statement changing a global variable, or the de-referenced value of a formal or global parameters since they may point to a global variable depending on the caller context.

6. PATH CONSTRUCTION AND VULNERABILITY ANALYSIS

In this section, we first introduce a path construction and analysis framework, and then detail accept path analysis and protocol state leakage analysis.

6.1 DFS Path Construction and Analysis Framework

The difference between an accept path and a leakage path merely lies in the analysis sink definition and the constraint analyzing and filtering rules that can be applied to reduce FPs. Thus, both analysis can be supported by a general path construction framework following a DFS (depth-first search) paradigm based on the path summary. As mentioned in §5.2, each path in a path summary has a constraint path part and a path end point part. Starting from an entry function, the DFS path construction process analyzes the paths in the summary, passes the paths to the callee functions if the path end point is a "function pointer" and continues the DFS process. The process ends when it reaches the analysis sink defined by an analysis task, and output concatenated inter-procedure paths. Like the inter-procedure propagation in taint analysis engine, here we need to use the calling context stored in the "function pointer" and $Inst(.)$ to change the variable context.

Path analysis with implicit flow tracking. In the path construction process, we analyze each path in the path summary following

the procedure shown in Fig. 4. We first check whether the path end point is the analysis sink or whether it is a "function pointer" that can call into the analysis sink. If not, this path is unrelated to the analysis task and we discard this path. After that, the property of the path is checked according to the purpose of the analysis task. If its property is considered valid for the analysis, it will be further judged on its importance; otherwise it is discarded. If its property is considered important and the layered analysis mode is on, the path result will be output. Otherwise, the DFS process continues to its callee function. The layered analysis mode will be described later in this section. When reaching the analysis sink, we only output the path if it is considered important.

The path property is determined by analyzing the variables and variable taints of the constraints in the constraint path. These constraints are directly related to the analysis, which we call *direct constraints*. However, besides direct constraints there are also other important constraints that the analysis sink depends on. For example, in Fig. 2, the sequence number check on line 2 in `tcp_validate_incoming()` is one of the most important checks preventing off-path packet injection, but it is not the constraint that `accept_payload()` is control dependent on. This dependence is passed through the return value of `tcp_validate_incoming()` to the direct constraint on line 3 in `tcp_rcv_established()`. In order to find these indirect constraints, we use $Track(\langle v, t\rangle)$ in variable taint value set stored in the path summary, and if t includes implicit flow taints, we track its taint path to the indirect constraint that passes these taint values. Based on our taint tracking design, these indirect constraints can be found in an inter-procedure fashion.

Layered path construction. To ensure minimum FNs, the path pruning rules in our accept path and leakage path analysis prefer to be conservative. However, this conservativeness may lead to more FPs, causing heavy analysis overhead. This problem can be quite serious for us since our output is program paths and nested constraint can exponentially increase the path number. To mitigate this problem, PacketGuardian supports a layered analysis mode, which is included at the bottom of Fig. 4. In this mode, when the path is important, we stop the DFS process and output the partial results. With these partial results, tool users can filter out the paths that are not of interest as early as possible, and feed the rest back to the tool to continue the DFS. As shown in our evaluation later in §7, this can largely reduce both the number of unimportant output paths and the analysis time. To reduce manual effort, PacketGuardian only stops when the path is considered important as this indicates that some constraints on the path are tightly related to the analysis but it is hard to automatically tell whether they are of interest.

6.2 Accept Path Analysis

In accept path analysis, the path is constructed and analyzed with attacker-controlled data and accept functions as input. Attacker-controlled data is usually the function parameters related the incoming packet (*e.g.,* `skb` in Fig. 2), and accept functions are functions that indicate the acceptance of the incoming packet, for example copying data to upper layers, or terminating the channel. If it is hard to find such functions, PacketGuardian also supports adding pseudo accept functions to label the analysis sink of interest.

Analysis sink check. In this analysis the analysis sink is a function, so we only consider the paths with end points of functions in path summary. Also, we only care about end point functions that are or may call into the accept functions. Thus, before the analysis, we first create a list of such functions by a DFS crawling process, and then in the analysis sink check discard the paths without an end point function in the list.

Constraint path property check. In the path analysis, each constraint is determined with a property of *protocol state check*, *weak check*, and *strong check*. For a constraint ct, we first check whether it is tainted by attacker-controlled data by looking at T_{exp}, and if not, it is a comparison related to a protocol state and thus labeled as channel state check. If it is tainted through explicit flows, we find out which variable v is attacker-controlled using the variable taint set in ct, and use exp to understand the comparison this constraint does for v. If it is tainted through implicit flows, the important comparison is done in a indirect constraint and we use the tracking described in §6.1 to find it out. We only consider this constraint to be weak check if (1) except v, all other variables are constants, or (2) this constraint requires v to be non-equal to non-constant variables. For the former, an attacker can easily spoof the corresponding packet fields to pass the check, and for the latter, it is very likely that a random value can pass the check. For all other cases, we conservatively label the constraint as strong check to avoid FNs.

In the path construction framework, if the path has a strong check constraint, it is considered important, and otherwise unimportant. A path is considered invalid if it has conflict constraints, *e.g.,* one constraint requires v to be larger than a value while another one requires it to be smaller. In our tool, we use a simple approach to detect this conflict by checking whether two constraints are exactly the same but one has $br = true$ and another has $br = false$.

Weak path candidate output. After the DFS path construction, all the output paths are valid accept paths. To reduce analysis effort, by default the path output consists of only protocol state check and strong check constraints. We include protocol state checks as it can help understand the channel conditions for an accept path. Note that we filter out the weak check constraints only in the last step so that the user can also configure the tool to show all constraints.

Since the goal is to identify the weakest accept path, we also apply path filtering to filter out stronger paths before the final output. If the constraints of one path is a subset of that of another path, the latter is stronger and will be filtered out.

6.3 Leakage Path Analysis

In this analysis, the information sources are the protocol states the strong accept path checks depend on, and the sinks are the channels accessible to an off-path attacker. Based on our path summary design, our sinks can be a function, a statement, or the paths reaching an important program point. This can support storage channels related to a statement that changes a global value [10, 24, 37], timing channels related to a statement or program path lengths [3, 29], or public events related to a function such as sending a packet [18].

Leakage detection. The taint summary for the entry function is a summarized variable tainting relationship, and we can directly tell whether there is possible storage channel leakage by checking if the storage channel sink variables are tainted through explicit or implicit flows. This is a convenient way to quickly tell the leakage status, but lacks detailed information for understanding the leakage, especially for implicit information leaks. Also, it cannot cover channels except storage channels. Thus, we also use the DFS path construction framework to construct leakage paths in this analysis.

Explicit information leaks are relatively easy to understand and PacketGuardian user can just use the taint propagation line numbers in the tracking summary to analyze the leakage. The user can also use the DFS path construction framework to construct the paths just like the accept path construction in §6.2 with a change of the analysis sink. However, in a protocol implementation the protocol states are usually not directly leaked through explicit flows to a storage channel – more common leakage is implicit information leaks as shown in recent vulnerability reports [18, 37].

For implicit information leaks, as discussed in §3.2, even though classic ones are generally considered of less value and commonly excluded in taint analysis tool design [4, 17, 21, 25], *attacker-controlled implicit information leaks* proposed by this paper are highly-exploitable according to existing vulnerability reports for practical protocol state leakage [18, 37]. Thus, our leakage path analysis targets this special type of leaks, and a very important benefit of this is that this can largely reduce FPs, which is a critical problem for implicit flow analysis [28].

In this following part of this section, we describe how to use the DFS path construction framework to find leakage paths through attacker-controlled implicit flows.

Analysis sink check. In this analysis, we filter out the paths which cannot reach the leakage sinks we defined. For storage channels, we can use the taint summary to check this, and for function sinks and other statement sinks, a DFS process like in the accept path analysis can be used to label function callees of interest, and discard the path of no interest in the DFS path construction.

Constraint property check. In the path analysis, each constraint is determined with a property of *unrelated*, *valid low entropy*, *invalid low entropy*, and *high entropy*. Since we target attacker-controlled implicit information leaks, the constraint is of interest only if it is tainted by both attacker-controlled data and the information sources. If not, it is labeled as unrelated constraint. If related, we find out the variables tainted by attacker-controlled data v^a and those tainted by the information source v^s respectively in direct and indirect constraints. With exp in the constraint we can figure out the comparison it does, and label the constraint as invalid low entropy if the constraint requires v^a to be equal to v^s, and as valid low entropy if the requirement for v^a is to be non-equal to v^s. For both cases the constraint has low entropy, but for the former it is unlikely to pass this check while for the latter it is very likely. For all other cases, we label the constraint as high entropy.

If the path has an invalid low entropy constraint, it is considered invalid and will be discarded. Otherwise, if it has a high entropy constraint, it is considered important. For all other cases it is considered unimportant. Like in accept path analysis, we also check the constraint conflicts and discard the paths with conflicts. For this analysis, layered analysis mode can be very helpful since it is usually hard to judge the entropy automatically. For example, in the illustrative example the receive and send window ranges are depending on dynamic protocol states and protocol design, making automatic judgement difficult. As shown in our evaluation in §7, with tool users filtering out paths with invalid low entropy constraints which are labeled conservatively as high entropy ones, finding practical vulnerabilities can be much more efficient.

Leakage path candidate output. Each leak is categorized by the high entropy constraints and the leakage sink, and by default PacketGuardian does not present unrelated and valid low entropy constraints to the user. PacketGuardian users can also configure the tool to output all constraints for more details. For an output path to break the non-inference property [20] and cause leakage, the same sink cannot be triggered for both true and false branches of a high-entropy constraint under all conditions. To check this, for a leakage path p_1 we first find all paths, say p_2, sharing the same sink with p_1 but takes the opposite branch in the high-entropy constraint ct^{high} in p_1. Then, we check whether all constraints in p_2 excluding ct^{high} are a subset of all other constraints in p_1. If so, p_1 is considered invalid and won't be included in the output.

7. EVALUATION

Following the design, we implemented the taint-based summarizer and vulnerability analyzer in OCaml with roughly 15K and 2.8K lines of code respectively. In this section, we evaluate the tool's effectiveness, accuracy, efficiency by applying it to 6 real network protocol implementations, covering 4 different network protocols. All experiments are run on a desktop computer with a 2.60GHz 8-core Intel Xeon CPU and 128 GB memory.

Code bases. The first code base we target is TCP in Linux kernel version 3.15.8, and we denote it as *TCP-Kernel*. Different from previous work which reported vulnerabilities in TCP code base by manual inspection [38], our tool performs automated analysis, and outputs not only all existing ones but also 11 new highly-exploitable ones. Besides TCP, we also choose two other famous protocols in the Linux kernel, SCTP and DCCP, denoted as *SCTP-Kernel* and *DCCP-Kernel*. Both of them are transport layer protocols providing reliable message delivery like TCP but having distinct features to support other communication requirements.

Besides transport protocols, we also analyze an application layer protocol, RTP, which is one of the most popular protocol for delivering audio and video over IP networks. We pick 3 different popular libraries, oRTP 0.24.1, PJSIP 2.4, and VLC 2.2.0, all of which implement RTP. In the following sections, we denote them as *RTP-oRTP*, *RTP-PJSIP*, and *RTP-VLC*.

For all 6 cases, the analysis chooses the function handling incoming packets as the entry, which are listed in Table 3. The last column shows the number of functions reachable from the entry point, showing the complexity of the code bases.

7.1 Tool Effectiveness and Accuracy

Table 5 summarizes the vulnerability and accuracy result for all 6 code bases. Column 2 describes the type of accept path defined in the analysis task, which in our experiments we consider 2 types: data and close, which means the analysis sink is to feed data to upper layers and to close the channel respectively. We call them *inject-payload* and *close-channel* accept paths in this section. Column 4–6 show the number of output paths, true positive (TP) number and false positive (FP) number. Here the ground truth is the feasible paths among all accept paths before pruning, and since our design is conservative in path pruning and filtering, we do not have any false negative (FN) cases for all 6 code bases. Column 3 shows the path number without the path pruning described in §6.2. As shown, our pruning reduces 42.6% output paths on average without introducing FNs. Since this output will be analyzed by an analyst, this pruning greatly reduces human efforts.

Column 7 shows the worst case number of packets needed for one injection after the accept path analysis, which is N_{pkt} defined earlier in §3. As shown, the N_{pkt} for 3 Linux kernel code bases is at least 10^7 for either inject-packet or close-channel cases, which are unlikely to be exploitable in practice. Their protections solely rely on a few "secret" protocol states unknown to the off-path attacker, which are listed in the last column.

In contrast, the 3 code bases for RTP protocol show diverse results. RTP-oRTP and RTP-PJSIP only need 51 and 3 packets to achieve injection, which are both easy to exploit in practice. But for RTP-VLC 2^{32} packets are needed, which is rather robust. All 3 code bases claim to follow RTP RFC 3550, but our result indicates that even following the same design, their packet injection robustness can be very different due to implementation differences.

For the code bases that do not have practical vulnerabilities in accept path analysis, we proceed to the second analysis step — protocol state leakage analysis. The N_{pkt} after leveraging leakage are shown in the column 8. For TCP, both protocol state `rcv_nxt` and `snd_nxt`/`snd_una` have high-entropy leakage, and largely degrade the N_{pkt} to only 64 and 32 for inject-payload and close-channel cases respectively. Leakage for `snd_nxt` and

Code base	Analysis entry function	Func #
TCP-Kernel	tcp_rcv_established()	1730
RTP-oRTP	rtp_process_incoming_packet()	141
RTP-PJSIP	on_rx_rtp()	67
RTP-VLC	rtp_queue()	22
SCTP-Kernel	sctp_sf_eat_data_6_2()	290
	sctp_sf_do_9_1_abort()	277
DCCP-Kernel	dccp_rcv_established()	359

Table 3: Statistics for the 6 code bases in our evaluation.

Tool w/o features	TP #	FP #	FN #	Low-entropy #
w/o field	4	501	0	27
w/o implicit flow	0	N/A	4	N/A
w/o pointer analysis	0	N/A	4	N/A
w/o layered	4	0	0	1336
w/ all above	4	0	0 (base line)	14

Table 4: Evaluation of accumulative improvement using *rcv_nxt* leakage in TCP-Kernel.

Code base	Type	Weak path output				Pkt # needed for injection	Pkt # needed for injection w/ channel state leakage	Protocol states the strong checks relying on
		Path # w/o pruning	Path #	TP #	FP #			
TCP-Kernel	Data	64	9	9	0	$(\frac{2^{32}}{win1} \times \frac{2^{32}}{win2})$ *	$(32+32)$	rcv_nxt, snd_nxt/una
	Close	40	1	1	0	2^{32}	32	rcv_nxt
RTP-oRTP	Data	21	15	10	5	51 *	N/A	N/A
RTP-PJSIP	Data	1	1	1	0	3	N/A	N/A
RTP-VLC	Data	32	8	4	4	2^{32} *	2^{32}	ssrc
SCTP-Kernel	Data	12	5	4	1	$2^{32} \times \frac{2^{32}}{rem\_win}$	$2^{32} + \frac{2^{32}}{rem\_win}$	my_vtag, base_tsn, cumulative_tsn_ack_point
	Close	5	2	2	0	2^{31}	2^{31}	my_vtag, peer_vtag
DCCP-Kernel	Data/Close	2	1	1	0	$\frac{2^{48}}{seqno\_win}$	$\frac{2^{48}}{seqno\_win}$	dccps_gsr/swh/swl

Table 5: Summary of vulnerability analysis results. Number labeled with "*" indicates that it can be smaller under special channel conditions. $win1$ and $win2$ is usually between 2^{14} to 2^{20}, rem_win is less than 4096 by default, and $seqno\_win$ is 100 during default initialization.

`snd_nxt`/`snd_una` have been reported previously [38] by manual discovery, and it is noteworthy that the `snd_nxt`/`snd_una` leakage has already been strengthened after Linux kernel version 3.8 and thus the vulnerability no longer exists. However, using our tool, we automatically find 4 high-entropy leakage for `rcv_nxt`, including the one reported before and 3 new ones. We validated all of them through experiments and confirm that they are indeed exploitable. For `snd_nxt`/`snd_una`, even after the fix, our tool successfully reports 13 new ones and 7 of them are validated.

For inject-payload case in SCTP-Kernel, a low-entropy leakage of `my_vtag` exists and also greatly reduces N_{pkt} from $2^{32} \times \frac{2^{32}}{rem\_win}$ to $2^{32} + \frac{2^{32}}{rem\_win}$. However, it is still a large number and not exploitable in practice. For RTP-VLC, DCCP-Kernel and close-channel case in SCTP-Kernel, no high-entropy leakage is output and thus their N_{pkt} with leakage remains the same. In §7.3, we provide more details on these results.

We further conduct an experiment to understand the effects of our static analysis enhancement. As shown in Table 4, we breakdown the accuracy improvement with each analysis enhancement using the `rcv_nxt` leakage analysis in TCP-kernel. The evaluation includes TP, FP, FN, and low-entropy leakage, and due to the difficulty of determining ground truth, we use the result of the tool with all features as baseline to evaluate FN for other cases. The results show that all static analysis enhancements, especially implicit flow tainting tracking, are necessary and play an important role.

7.2 Tool Efficiency

Before the taint analysis, the code pre-process is a one-time effort which takes around 8.7 hours for the entire Linux kernel, and only less than a minute for oRTP, PJSIP, and VLC.

For taint-based summarizer, since summarizing the entire Linux kernel is infeasible, we limit the scope of TCP-Kernel, SCTP-Kernel, and DCCP-Kernel to the `net` folder under the self-contained Linux kernel networking subsystem. TCP-kernel takes the longest time of 7.8 hours, which we believe is acceptable considering that the computed summary can be reused later for further analysis. In addition, the time can further improved by analyzing functions in parallel as shown in Saturn [52], which is another advantage of our choice of summary-based approach.

With the function summaries, the accept path and protocol state leakage path analysis are very efficient, and perform these analysis on all code bases is less than 10 seconds. Note that this efficiency also benefits a lot from our layered analysis mode, for example, for `rcv_nxt` leakage analysis in TCP-Kernel, it takes 984.5 seconds in total if not using layered analysis mode.

7.3 Result analysis

In this section, we detail the vulnerability analysis results summarized in Table 5. Due to the space limit we cannot provide code-level details for all results, and for more details about the experiment setup and vulnerability results, please visit our result website **http://tinyurl.com/PacketInjectionVulnerability** [1].

7.3.1 TCP-Kernel

Accept path analysis. Our tool outputs 9 inject-payload accept paths which are all TPs. 6 out of them are in TCP fast path processing. The conditions for entering fast path is shown in Fig. 6. On line 1, to match the prediction flag it requires the receiver's exact send window size, which is possible to achieve in some cases, *e.g.,* when TCP connection is idle. The hard requirement of falling into fast path is that the sequence number, `seq`, needs to equal to the protocol state `rcv_nxt` on line 2. The other 3 output paths are on the slow path, which correctly implements the latest standard specified in RFC 5961 to defend against off-path attacks. In short, they all require the `seq` to fall in the receive window, and `ack` to fall into another window like shown in Fig. 2. Thus, their N_{pkt} is roughly $\frac{2^{32}}{win1} \times \frac{2^{32}}{win2}$.

For channel-close case, our tool outputs 1 path due to the effectiveness of our pruning and it is a TP. This path resets the TCP connection in `tcp_validata_incoming()`, and requires `seq` to be equal to `rcv_nxt`. Thus, its N_{pkt} is 2^{32}. Note this an update as specified in RFC 5961 from the previous TCP implementation where a TCP RST is accepted as long as the `seq` falls in the receive

Code base	Protocol state	Output #	Validated #	Hard to trigger #	FP #	Low-entropy #
TCP-Kernel	rcv_nxt	18	4	0	0	14
	snd_nxt/una	65	7	6	9	43
SCTP-Kernel	base_tsn, cumulative _tsn_ack_point	3	N/A	N/A	0	3
DCCP-Kernel	dccps_gsr/swh/swl	5	N/A	N/A	1	4

Table 6: Protocol state leakage analysis result. Ssrc for RTP-VLC and my/peer_vtag for SCTP-Kernel is not included since our tool does not output any high-entropy leakage for them.

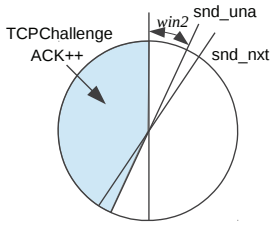

Figure 5: Leakage of snd_nxt through sink `TCPChallengeACK`.

```
1: If ((tcp_flag_word(th) & TCP_HP_HITS) == tp->pred_flags &&
2:     TCP_SKB_CB(skb)->seq == tp->rcv_nxt &&
3:     !after(TCP_SKB_CB(skb)->ack_seq, tp->snd_nxt) {
     ...
4: }
```

Figure 6: Code snippet for conditions of entering TCP fast path.

window. This change significantly increases the blind in-window RST attacks.

Protocol state leakage. Both inject-payload and close-channel accept paths are protected by protocol state `rcv_nxt`, so we first use this as the leakage source in our leakage path analysis. In our experiments, we use network statistics output in `netstat`, `snmp` and `sockstat` in `/proc/net/` as storage channel leakage sinks. To find the variables that are output to these sinks for taint analysis, we perform static analysis in file `net/ipv4/proc.c` starting from `proc_create()`, locate the proc file function operation registration and find the target variables in the output function, *e.g.*,, `netstat_seq_show()` for `netstat`. With these sink variables, we first check taint summary for the entry function, and find that these variables are tainted only by `rcv_nxt` through implicit flow. Then we use these tainted variables as leakage sinks in the leakage path analysis.

The leakage results are summarized in Table 6. For `rcv_nxt` our tool outputs 18 leakage candidates, and 4 of them are high-entropy TPs. 14 of them are low-entropy leakage, which are mostly pruned out by layered analysis. Note that since falling into fast path requires `seq` equaling to `rcv_nxt`, all fast path related leakage are filtered out automatically as invalid low-entropy leakage. Among the 4 TPs, one of them is reported by previous work [38] by manual discovery. The other 3 are all new discovery, and one uses the same high-entropy constraint in `tcp_send_dupack()` as the one reported, but has a newly-discovered sink *TCPDSACK-OldSent* in `netstat`. For the other 2, the attacker packet also makes the code calling into `tcp_send_dupack()` but with different calling context by deliberately failing the PAWS check, *e.g.,* by using an old time stamp, before the `seq` check (line 2 in Fig. 2).

After knowing `rcv_nxt`, the attacker can successfully reset the connection and causing DoS. However, to injection payload, the attacker still lacks the knowledge of `snd_nxt` or `snd_una` to pass the `ack` check. We then run another leakage path analysis with these two values together as leakage sources. Like `rcv_nxt`, the sink variables are only tainted by implicit flow. In this setting, we assume that the attacker already knows the correct `rcv_nxt` using the leaks discovered above. For leakage sinks, we use the same ones as those in the `rcv_nxt` analysis.

Since at this time the exact `rcv_nxt` is known, the attacker's packet can exploit leakage vulnerabilities in more program paths including both fast path and slow path area. Our tool outputs 66 leakage candidates for `snd_una`, and 43 of them are low-entropy leakage which are filtered during layered analysis. Among the 9 FPs, 3 cases are caused by requiring packet length to be smaller than data offset field or having an incorrect checksum value, but actually such packets are dropped in `tcp_v4_rcv()` before entering our entry function `tcp_rcv_established()`. Other 4 FP cases requires a fast path protocol state `tcp_header_len` to be greater than 4, but in the implementation it can only be 0 or 4. The last 2 cases are caused by conflicting constraints across procedures, which can be solved by applying more advanced constraint solver such as a SMT solver [13], which we leave as future improvement. The 13 TPs are all new discovery, and 8 are in fast path while 5 are in slow path. All the 8 fast path ones use the comparison between `snd_nxt` and `ack` one line 3 in Fig. 6, and after this comparison, there are 8 different sinks in `tcp_rcv_established()`, `tcp_send_ack()`, *etc.*

The 5 leaks in slow path both goes into `tcp_ack()`, and the high-entropy constraint they use is on line 1 and 2 in `tcp_ack()` of Fig. 2. As shown in Fig. 5, probability of reaching the `return` on line 3 is $\frac{2^{31}-win2}{2^{32}}$, which leaks around 1 bit of information under the assumption that $win2$ is usually smaller than 2^{20}. In the code base, right before the `return` on line 3 there is a `tcp_send_challenge_ack()`, in which sink *TCPChallengeACK* is triggered when the challenge ACKs that are already sent is under a threshold set in `/proc/sys/net/ipv4/tcp_challenge_ack_limit`, which is usually around 100.

Validation. We setup a TCP connection between desktop computer A and B, and have another attack computer using raw socket to send attack packets to B to validate these leaks. Computer B is installed with Linux kernel 3.15.8 and added debug information along the program path to validate whether the leakage path is triggered exactly as our tool output, and at the same time monitor the corresponding leakage sinks in A's proc file system. For `rcv_nxt`, we validate all 4 high-entropy leakage. For `snd_nxt`/`snd_una`, 7 out of the 13 cases are validated. The other 6 cases are relatively hard to trigger, for example, 5 of them requires kernel configuration `CONFIG_NET_DMA`, which is only available for processors of certain architecture, *e.g.,* Intel Xscale I/O processors 32/33x.

7.3.2 RTP

Since the 3 RTP code bases flow the same network protocol and thus similar to each other in most of the core logic, we cover their results all together in this section. RTP usually doesn't have the option to close the channel with an incoming packet, so our accept path analysis are all inject-payload accept path analysis.

RTP-oRTP. The output for RTP-oRTP has 15 paths, among which 10 are TPs and 5 are FPs. The 5 FPs are all caused by two channel variables having the same meaning, one indicating whether `ssrc` is set, and another indicating whether the first packet is delivered. The changing of two variables is usually correlated and thus they have equivalent values, but our analysis treats them separately, resulting in FP paths with semantically-conflicting constraints. Among the 10 TP cases, 3 requires guessing the correct

```
1: matched_ssrc = NULL;
2: for (i=0; i<n; i++) {
3:    if (pkt->ssrc == ssrc) {
4:      matched_ssrc = ssrc;
5:      break;
6:    }
7: }
8: if (matched_ssrc != NULL) {
9: ...
```

Figure 7: False positive causes for RTP-VLC accept path analysis.

32-bit protocol state `ssrc` value, thus N_{pkt} is 2^{32}. However, another 3 TPs indicate that in its logic by default after 50 packets with a new `ssrc` and consecutive sequence numbers, RTP-oRTP will change the `ssrc` to the new one, making the N_{pkt} reducing to 51. The other 6 TPs are all under very special channel conditions, for example N_{pkt} can be as low as 1 if the attacker precisely captures the moment when `ssrc` is not set yet.

RTP-PJSIP. For RTP-PJSIP, the output only has 1 path and it is a TP. In this path, unlike RTP-oRTP, it changes its protocol state `ssrc` right away if it sees a new one, and relies its robustness solely on the sequence number. According to its logic output by our tool, 2 packets with consecutive sequence numbers will trigger a channel restart, and the 3rd packet's payload will be accepted. Thus, N_{pkt} for RTP-PJSIP is 3.

RTP-VLC. The output for RTP-VLC has 8 paths and 4 of them are TPs. The causes of the 4 FPs are shown in Fig. 7. In these paths, it takes both the false branch of `i<n` on line 2 and the true branch on line 8, which is actually not feasible. This is mainly because we construct paths in a flow-sensitive framework and merge the paths from the `break` on line 5 and `i<n` on line 2 when reaching line 8. This can be solved by path-sensitive analysis which has higher precision but also much higher overhead. For the 4 TPs, 2 of them requires the correct `ssrc`, thus their N_{pkt} is 2^{32}. Like RTP-PJSIP, the other 2 TPs change `ssrc` right away, and since RTP-VLC maintains sequence number state separately for each `ssrc`, the N_{pkt} is actually 1. However, changing `ssrc` in RTP-VLC is only when the channel is configured to support more than one `ssrc`, and by default RTP-VLC only supports one. Thus, in normal cases the N_{pkt} is 2^{32} for RTP-VLC.

RTP-VLC protocol state leakage. Among these 3 RTP code bases, only RTP-VLC is hard to inject in default setting due to the protection from the protocol state `ssrc`. In the taint summary of the entry function, 14 variables are tainted by `ssrc`, all through implicit flows. To check the leakage possibility, we set all these 14 variables as leakage sinks in the leakage path analysis but no high-entropy leak is found.

Validation. We build oRTP 0.24.1 and PJSIP in pjproject 2.4, establish audio communication between computer A and B, and read payload in B from application layer APIs. Since proc file `netstat` only shows the local IP address and UDP port for the RTP channel, the attacker computer sends attack RTP packets to B with correct destination IP address and port but different source IP address and port from A's. In the audio data we sent, we include packet number so that we know which packet's payload gets in to the upper layer. We successfully validate that the payload of the 51-st packet for oRTP, and the 3rd packet for PJSIP gets accepted. We also confirm that for VLC without correct `ssrc` the injection cannot succeed.

7.3.3 SCTP-Kernel

Accept path analysis. Our tool outputs 5 paths for inject-payload accept path analysis, and 4 are TPs. One SCTP packet can have multiple chunks, and the 1 FP case is because it requires previous chunks from the same packet to have ready been accepted, which is an implementation semantic information that is not known by our tool. One of the TPs has no special channel condition dependence, and it requires (1) it has the correct 32-bit protocol state `my_vtag`, and (2) the sequence number `tsn` falls into a window win starting from a protocol state `base_tsn`, and by default this win is 4096. At the same time, `tsn` also needs to be larger than the previously-received `tsn`, stored in a third protocol state, `cumulative_tsn_ack_point`. We denote the valid tsn range as rem_win, which is win excluding the parts before `cumulative_tsn_ack_point`. Thus, the N_{pkt} is $2^{32} \times \frac{2^{32}}{rem\_win}$. The other 3 TPs all depend on special channel conditions and their N_{pkt} is not smaller.

For close-channel case, our tool outputs 2 results and both are TPs. One path handles error cause code in the incoming packet, and the other handles packets without error cause code. In both cases, the packet needs to have correct `my_vtag` or `peer_vtag`, which are both 32 bits. Considering the probability that `my_vtag` equals `peer_vtag`, the N_{pkt} is 2^{31}.

Protocol state leakage. The accept paths are protected by `my_vtag`, `base_tsn`, and `cumulative_tsn_ack_point`, so we use them as leak sources. For sinks, we also use storage channel like in TCP-Kernel, and for SCTP we use SNMP statistics in proc file `/proc/net/sctp/snmp`. To get the variables in these sinks, we perform the same static analysis described in §7.3.1.

We run the leakage path analysis and find no high-entropy leaks. From the analysis log we find that all leakage paths start with `my_vtag` check, and thus are low-entropy leaks. One of them can be used to tell whether the attack packet has the correct `my_vtag` by looking at sink *SctpInPktDiscards*. This needs 2^{32} packets in the worst case, but it can still be helpful to lower the N_{pkt} from $2^{32} \times \frac{2^{32}}{rem\_win}$ to $2^{32} + \frac{2^{32}}{rem\_win}$.

With the knowledge of `my_vtag`, we still needs to have a `tsn` that can fall into the window specified by `base_tsn` and `cumulative_tsn_ack_point`. We use them as leak sources and find 3 leaks but all are low-entropy ones.

For the close-channel accept paths, the protocol states they rely on are `my_vtag` and `peer_vtag`. In the taint summary, the sinks are also only tainted by implicit flows, but our tool outputs no high-entropy leaks for both of the sources.

7.3.4 DCCP-Kernel

Accept path analysis. In DCCP, the checks for copying payload and resetting connection are the same. In this analysis, our tool outputs 1 path and it is a TP. In this path, the DCCP sequence number `seqno` needs to fall into a sequence window $seqno\_win$ around a protocol state `dccps_gsr` as long as 48 bits, and the higher and lower bounds of this window are another two protocol states `dccps_swh` and `dccps_swl`. Thus, the N_{pkt} is $\frac{2^{48}}{seqno\_win}$. Note that the initial size of this seq_win is only 100, making it impractical to inject. In normal cases there should be another check for the DCCP acknowledge sequence number `ackno`, but as shown in our analysis output, attacker can send a DATA type DCCP message without acknowledge sequence number to avoid that check.

Protocol state leakage. We use all 3 protocol states as sources in this analysis. For sinks, currently DCCP does not create a proc file to store global statistics yet, but it does have a structure for SNMP statistics like TCP-Kernel and SCTP-Kernel, which has same leakage potential if enabled in the future. Thus, we use these variables as leakage sinks. Our tool outputs 5 leaks and 4 of them are TPs. The 1 FP path requires (1) the attack packet is a `SYNC`

or SYNCACK packet having the right ackno, (2) seqno is larger than dccps_swl, and (3) it fails the *seqno_win* check. However, when (1) and (2) happen, dccps_gsr is updated with seqno and it won't fail the *seqno_win* check. In our analysis, we can know that dccps_gsr is updated, but cannot be sure that seqno can pass the *seqno_win* check. The 4 TPs all require seqno to fall into *seqno_win*, and thus are all low-entropy leaks.

8. LIMITATION AND FUTURE WORK

Possible FNs due to implementation simplification. We design and implement a high precision data flow analysis with implicit flow tainting and pointer analysis to avoid FNs as much as possible. However, there may still be cases causing FNs due to simplified implementation. For example, as described in §5.1, we add an iteration limit of loops to avoid adding recursive fields and this may lead to FN cases if the leakage sinks have recursive fields.

Failure to identify semantically-conflicting and low-entropy constraints. As discussed in §7.3, the majority of the FPs are caused by conflicting constraints that are tricky to identify. In the future, we plan to use a SMT solver [13] commonly employed by symbolic execution as tool improvement.

Limited scope of storage channel. As described in §6.3, our tool is designed with the capability to cover a range of leakage channels such as storage channels, data timing channels, and public events like sending packets. However, in our experiments we only use storage channels in proc file system as leakage sinks, and may miss practical vulnerabilities leaked through other channels. In the future, we plan to incorporate other sinks in the leakage path analysis.

9. RELATED WORK

Network protocol analysis. To detect protocol design vulnerabilities, prior work has used formal methods such as model checking and specification languages to perform rigorous protocol specification testing [6, 7]. However, these cannot prevent vulnerabilities due to weak implementations. For implementation vulnerabilities, static analysis has been applied to identify system DoS vulnerabilities [8], protocol manipulation attacks [30], and interoperability problems [36]. However, none focused on packet injection vulnerability causing payload injection or network DoS.

Static analysis for taint-style vulnerability. For taint-style vulnerabilities, static analysis tools have been designed to detect buffer overflow [12], format string vulnerabilities [45], and SQL injection and XSS [25, 27, 48]. Recently, Yamaguchi *et al.* [53] propose to use code property graph to effectively mine such vulnerabilities in large amounts of code. Different from them, our analysis targets packet injection instead of code injection, which requires handling much more and also diverse checks due to header field semantics. Moreover, we have a follow-up leakage analysis which is not included in previous tools.

Static taint analysis are also used to detect information leakage vulnerabilities in recent years, especially for privacy leakage in Android system [4, 16, 17, 21]. However, these tools exclude implicit flow tainting due to its low-entropy in leakage and the problem of high FPs [28]. In comparison, our tool taints implicit flows as required by our analysis goal, and proposes to target attacker-controlled implicit information leaks to mitigate the FP problem while maintaining high accuracy.

Side-channel attack and detection. Recently years witness a rise in side-channel attack discoveries. For **storage channels**, proc file systems have been abused as side-channels to infer keystrokes [56], webpage [24], and system state [10]. In particular, Qian et al. [37, 38] used proc file packet counters to infer TCP sequence number. Another popular channel is **timing channel**, including code path [29], data [3], and cache-access timing channel [22, 54]. In network protocol attacks, some header fields are also found to be useful for inferring sequence number [18]. In comparison, our goal is not to report new side-channel attack but focuses on designing an automated tool to systematically detect side channels.

In contrast to side channel discovery, side channel detection has been less explored. Dynamic analysis such as black-box testing has been used to find side channels in web application [9], and timing side channels in SSL/TLS implementation [33]. To overcome the drawback of analysis completeness in dynamic analysis, static analysis tools are also developed to detect web application and cache side channels [14, 55]. Compared to them, our tool can detect storage side channels, which has not been covered in existing tools, and moreover, our target is protocol state leakage, which is different from previous work.

10. CONCLUSION

In this paper, we design and implement an effective and scalable static program analysis tool, PacketGuardian to systematically analyze the security properties of network protocol implementations against off-path packet injection attacks. PacketGuardian uses a context-, flow-, and field-sensitive taint analysis with pointer analysis to achieve high precision, and also targets attacker-controlled implicit information leaks. The solution significantly eases the classic problem of false positives of implicit flow tracking while still yields high detection accuracy of practical exploits. By applying our tool on 6 real network protocol implementations, we are able to discover new and realistic vulnerabilities confirmed by proof-of-concept attacks for both Linux kernel TCP and 2 out of 3 RTP implementations.

Acknowledgments

We would like to thank Danfeng Zhang, Rajiv Gupta, Jia Chen, Sanae Rosen, Jason Jong Kyu Park, Lingjia Tang, and the anonymous reviewers for providing valuable feedback on our work. This research was supported in part by the National Science Foundation under grants CNS-1318306 and CNS-1464410, as well as by the Office of Naval Research under grant N00014-14-1-0440.

11. REFERENCES

[1] Analysis result website. http://tinyurl.com/PacketInjectionVulnerability.

[2] STAC - Static Taint Analysis for C. http://code.google.com/p/tanalysis/.

[3] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On Subnormal Floating Point and Abnormal Timing. In *IEEE Symposium on Security and Privacy*, 2015.

[4] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *PLDI*, 2014.

[5] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict Control Dependence and its Effect on Dynamic Information Flow Analyses. In *ACM ISSTA*, 2010.

[6] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal Verification of Standards for Distance Vector Routing Protocols. *Journal of the ACM*, 2002.

[7] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous Specification and Conformance Testing Techniques for Network Protocols, as Applied to TCP, UDP, and Sockets. *SIGCOMM*, 2005.

[8] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *CSF*, 2009.

[9] P. Chapman and D. Evans. Automated Black-box Detection of Side-channel Vulnerabilities in Web Applications. In *CCS*, 2011.

[10] Q. A. Chen, Z. Qian, and Z. M. Mao. Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks. In *USENIX Security*, 2014.

[11] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *IEEE Symposium on Security and Privacy*, 2010.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security*, 1998.

[13] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.

[14] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Usenix Security*, 2013.

[15] E. Dumazet. Kernel discussion on ACK flag. `http://comments.gmane.org/gmane.linux.network/253369`, 2012.

[16] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling User-trigger Dependence for Android Malware Detection. *Computers & Security*, 49:255–273, 2015.

[17] C. Gibler, J. Crussell, J. Erickson, and H. Chen. AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale. In *TRUST*, 2012.

[18] Y. Gilad and A. Herzberg. Off-Path Attacking the Web. In *USENIX WOOT*, 2012.

[19] Y. Gilad and A. Herzberg. When tolerance causes weakness: the case of injection-friendly browsers. In *WWW*, 2013.

[20] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, 1982.

[21] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard. Information-flow Analysis of Android Applications in DroidSafe. In *NDSS*, 2015.

[22] D. Gullasch, E. Bangerter, and S. Krenn. Cache Games–Bringing Access-based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy*, 2011.

[23] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural Pointer Alias Analysis. *TOPLAS*, 21(4):848–894, 1999.

[24] S. Jana and V. Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy*, 2012.

[25] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy (SP)*, 2006.

[26] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *NDSS*, 2011.

[27] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE*, 2009.

[28] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't Live with 'em, Can't Live Without 'em. *Information Systems Security, Lecture Notes in Computer Science*, 5352:56–70, 2008.

[29] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

[30] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding Protocol Manipulation Attacks. In *SIGCOMM*, 2011.

[31] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*, 2015.

[32] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *NDSS*, 2011.

[33] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *USENIX Security*, 2014.

[34] B. Muller. Whitepaper: Improved DNS Spoofing Using Node Re-delegation. `https://www.sec-consult.com/fxdata/seccons/prod/downloads/whitepaper-dns-node-redelegation.pdf`.

[35] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C programs. In *CC*, 2002.

[36] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *NSDI*, 2015.

[37] Z. Qian and Z. M. Mao. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *IEEE Symposium on Security and Privacy*, 2012.

[38] Z. Qian, Z. M. Mao, and Y. Xie. Collaborative tcp sequence number inference attack: how to crack sequence number under a second. In *CCS*, 2012.

[39] Ramaiah, Anantha and Stewart, R and Dalal, Mitesh. Improving TCP's Robustness to Blind In-Window Attacks. rfc5961, 2010.

[40] A. Rane, C. Lin, and M. Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security*, 2015.

[41] S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS*, 2014.

[42] T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.

[43] A. Rountev, M. Sharp, and G. Xu. IDE Dataflow Analysis in the Presence of Large Object-oriented Libraries. In *CC*, 2008.

[44] M. Sagiv, T. Reps, and S. Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science*, 167(1):131–170, 1996.

[45] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security*, 2001.

[46] K. Smith-Strickland. We're Closer to an Encrypted Internet than You Think. `http://gizmodo.com/two/-thirds-of-internet-traffic-could-be-encrypted-by-ne-1702659626#`, May 2015.

[47] S. Son and V. Shmatikov. The Hitchhiker's Guide to DNS Cache Poisoning. In *Security and Privacy in Communication Networks*, pages 466–483. Springer, 2010.

[48] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.

[49] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, And Tools (2nd Edition)*. Addison Wesley, 2006.

[50] V. Varadarajan, T. Ristenpart, and M. Swift. Scheduler-based defenses against cross-vm side-channels. In *Usenix Security*, 2014.

[51] R. P. Wilson and M. S. Lam. Efficient Context-sensitive Pointer Analysis for C Programs. In *PLDI*, 1995.

[52] Y. Xie and A. Aiken. Saturn: A Scalable Framework for Error Detection using Boolean Satisfiability. *TOPLAS*, 2007.

[53] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *IEEE Symposium on Security and Privacy (SP)*, 2014.

[54] Y. Yarom and K. E. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. *USENIX Security*, 2014.

[55] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated Detection and Quantification of Side-channel Leaks in Web Application Development. In *CCS*, 2010.

[56] K. Zhang and X. Wang. Peeping Tom in the Neighborhood: Keystroke Eavesdropping on Multi-User Systems. In *USENIX Security*, 2009.

[57] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.

[58] Y. Zhang and M. K. Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *ACM CCS*, 2013.

[59] Y. Zheng and X. Zhang. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection. In *ICSE*, 2013.

[60] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets from Android Public Resources. In *CCS*, 2013.