# Sidebuster: Automated Detection and Quantification of Side-Channel Leaks in Web Application Development

Kehuan Zhang, Zhou Li, Rui Wang,
XiaoFeng Wang
School of Informatics and Computing,
Indiana University, Bloomington, IN, USA
{kehzhang, lizho, wang63,
xw7}@indiana.edu

Shuo Chen
Microsoft Research,
Microsoft Corporation,
Redmond, WA, USA
shuochen@microsoft.com

## ABSTRACT

A web application is a "two-part" program, with its components deployed both in the browser and in the web server. The communication between these two components inevitably leaks out the program's internal states to those eavesdropping on its web traffic, simply through the *side channel* features of the communication such as packet length and timing, even if the traffic is entirely encrypted. Our recent study shows that such side-channel leaks are both fundamental and realistic: a set of popular web applications are found to disclose highly sensitive user data such as one's family incomes, health profiles, investment secrets and more through their side channels. Our study also shows that an significant improvement of the current web-application development practice is necessary to mitigate this threat. To answer this urgent call, we present in this paper a suite of new techniques for automatic detection and quantification of side-channel leaks in web applications. Our approach, called *Sidebuster*, can automatically analyze an application's source code to detect its side channels and then perform a rerun test to assess the amount of information disclosed through such channels (quantified as the entropy loss). Sidebuster has been designed to work on event-driven applications and can effectively handle the AJAX GUI widgets used in most web applications. In our research, we implemented a prototype of our technique for analyzing GWT applications and evaluated it using complicated web applications. Our study shows that Sidebuster can effectively identify the side-channel leaks in these applications and assess their severity, with a small overhead.

## Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

## General Terms

Security

## Keywords

Side-channel leak detection and quantification, Web application, Program analysis

*CCS'10,* October 4–8, 2010, Chicago, Illinois, USA.
Copyright 2010 ACM 978-1-4503-0244-9/10/10 ...$10.00.

## 1. INTRODUCTION

A web application is "desktop software in the browser", which serves an end user through both its browser-side component and web-server-side component. Such web-based computing, also known as software-as-a-service (SaaS), has been increasingly used to replace desktop applications and is rapidly becoming mainstream. This fundamental change of computing paradigm, however, brings in new privacy threats, which, in some cases, have serious consequences. Our recent study [13] shows that even with the state-of-the-art cryptographic protection, the communication between a web application's client/server components is vulnerable to *side-channel attacks*, in which a network eavesdropper can infer the content of encrypted web traffic from its observable attributes such as packet sizes and sequences. As an example, consider a situation where one prepares her tax returns through an online tax application; though the web traffic generated by the software is protected by HTTPS, her adjusted gross income (AGI) can still be identified from the disparate traffic patterns associated with different execution paths, which are related to her eligibility for various tax credits.

**Side-channel leaks in web applications.** Although side-channel leaks in encrypted web traffic have long been known [38], the problem was found in our recent study to be inherent to web applications [13]: fundamentally, the client/server split of these applications exposes part of their internal information flows to a network eavesdropper, who can therefore deduce their state transitions and the user data that trigger these transitions. To make things worse, such data are often of low entropy and thus easy to enumerate and compare, thanks to the standard programming methods like AJAX [22], which query a server in response to even a single letter entered into a text box or an item selected from a list. Our findings [13] highlight the pervasiveness of this problem in popular web applications, including Google/Yahoo/Bing search engines, popular tax programs, online health information systems and mutual fund investment websites. The information being leaked out include one's health record (medications/surgeries/disease conditions), annual family incomes and investment secrets (mutual fund selections and allocations), even when web traffic is protected by HTTPS [7]. Also inferable are the query words corporate employees enter into their laptops, despite the protection of the up-to-date WPA/WPA2 Wi-Fi encryptions [8]. The causes of the problem were found to be the design features of these applications [13], for example, the popular auto-suggestion widget triggered by every keystroke and distinguishable execution paths determined by income thresholds defined in tax laws.

Mitigation of this side channel problem turns out to be very challenging [13]. The information leaks in individual applications are specific to their functionality, semantics and even domain knowledge (e.g., tax laws and public financial data). On the other hand, while general mitigation measures, such as packet padding, have been well studied, applying these techniques blindly without considering a web application's specific features often render them less effective and exceedingly costly. For example, to suppress the information leaks from a famous tax application without detecting its side-channel vulnerabilities, the communication overhead incurred by packet padding was found to be prohibitive [13]. The same problem is likely to happen to other universal fixes, such as producing constant-rate noise packets, simply because the web contents (e.g., images, videos, code, etc.) exchanged *within* a web application are usually highly diverse, and their traffic attributes are often too disparate to hide [13]. Fundamentally, given the complexity of state transitions within web applications and the diversity of their internal information communicated through the network, a general and effective defense can only be built upon in-depth analyses of individual applications. This calls for changes to the way today's web applications are built and tested, to add the steps for detecting and removing their side-channel vulnerabilities.

**Our work.** The first step of such a principled development methodology is to identify potential side-channel weaknesses from a web application and determine their gravity. This requires automatic program analysis technologies to be developed to support in-depth analysis of increasingly complicated web applications. In our research, we made the first step towards building such technologies. We present in this paper *Sidebuster*, the first approach for automatic detection and quantification of side-channel leaks in web applications. Based upon a set of "taint sources" the developer labels as sensitive, Sidebuster conducts an *information-flow analysis* [44] on source code to track the propagation of "tainted" data across a program's client/server components. Whenever the tainted data are found to be transmitted to the network through an encrypted channel, an information-leak evaluation is performed to understand whether the side-channel information of the channel, such as packet sizes and sequences, can be used to infer the content of the data. Whenever a branch condition is found to be tainted and its branches involve client/server communications, our tool evaluates whether the attributes of such communications reveal the sensitive condition. We also propose new techniques for analyzing GUI widgets, such as auto-suggestion lists, which are triggered by input events (e.g., letters being entered) to synthesize different user inputs into an integral variable (e.g., a query word) that the developer labels as a "taint target".

Actually, every web application gives away more or less information through its side-channels. However, not all such leaks deserve serious attentions and mitigation efforts. For example, people could live with disclosure of the lengths of their query words and certain operations they performed, such as sending/receiving emails. A question, therefore, becomes how to quantify the private information that can be inferred from a side-channel vulnerability. This question can be answered by a dynamic analysis, as the encrypted traffic of a web application is actually produced by its underlying web platform (web servers/browsers), whose source code is often beyond the access of the application developer. In this paper, we also describe our design of a quantification technique that systematically re-runs selected portions of a web application to understand how the domain of a taint source or target can be partitioned by its side-channel leaks. We also report an evaluation study that demonstrates the effectiveness of our techniques.

**Contributions.** Here we outline the contributions of this paper:

•*Side channel detection.* We propose the first technique for automatic detection of side-channel leaks in web applications. Built upon existing technologies such as taint analysis, our approach can effectively evaluate the source code of those applications to identify the program locations where sensitive user data affects encrypted communication through data flows and/or control flows. We offered novel solutions to the technical challenges associated with the special features of web applications, particularly their extensive use of AJAX GUI widgets.

•*Side channel quantification.* We present a novel technique for quantifying the side-channel leaks in web applications. The new technique can measure not only the information disclosed from a single taint source but also that aggregated from multiple sources, according to the dependency relations among these sources.

•*Implementation and evaluation.* We implemented our techniques into a prototype for analyzing the web applications built upon Google Web Toolkit (GWT) [18], and evaluated it over 6 real-world or synthesized applications. Our study shows that Sidebuster worked effectively on these applications and incurred acceptable overheads.

Although Sidebuster made an important first step toward mitigating the side channel threats in web applications, its current design is still preliminary. We cannot guarantee the completeness of our side-channel analysis. For example, our rerun test cannot exhaustively evaluate all possible combinations of sensitive inputs when the input space is large. Actually, a complete side-channel analysis is a daunting task, considering the complexity of the problem: not only could subtle information leaks hide deep inside myriad program structures, they could also be caused by the domain information specific to individual applications, such as the relations between diseases and their treatments [13]. This leads us to believe that instead of closing a case, our work actually opens a new episode. Continued research efforts are expected on this important yet understudied direction.

**Roadmap.** The rest of the paper is organized as follows. Section 2 presents an overview of Sidebuster and the assumptions made in our research. Section 3 and 4 elaborates our designs and implementations of side-channel detection/quantification techniques respectively. Section 5 reports our experimental study of these techniques. Section 6 discusses the limitations of our approach and potential future research. Section 7 reviews related prior research and Section 8 concludes the paper.

## 2. OVERVIEW

In this section, we first discuss the backgrounds of our research, then explain how Sidebuster detects and quantifies side-channel leaks in web applications through an example. The designs of these techniques are elaborated in Section 3 and 4. We also implemented a prototype for analyzing the web applications built upon GWT [18] (GWT is a widely used web application platform that automatically converts a Java program into a JavaScript-based client component and a Java Servlet on the server side).

### 2.1 Adversary Model and Research Problem

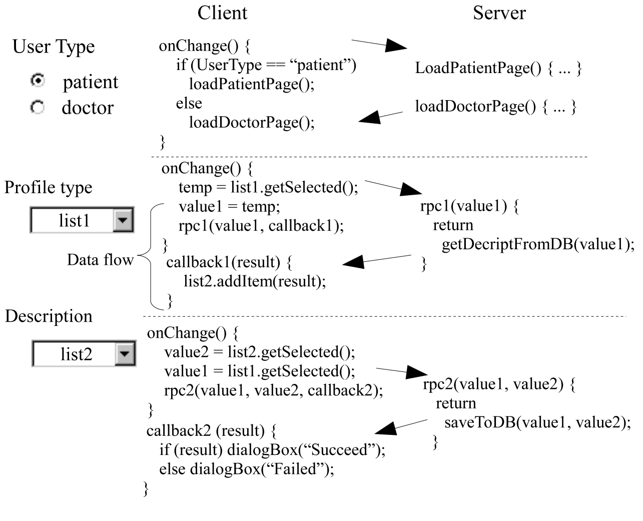
Similar to a traditional desktop application, a web application has its control flows and data flows. The difference is that a subset of such information flows go through the network and as a result, are subject to network eavesdropping. In our research, we consider an adversary who intends to infer the content of the *encrypted* communication between the application's client and server components. The adversary does not have a direct observation of the content but can only eavesdrop on encrypted web traffic to glean the traffic features (indirectly related to the content), such as packet lengths and sequences. We also assume that the adversary has a test account on the target application, so that he/she can study the traffic features of the test account before analyzing the traffic of the victim user's account. Our research aims at mitigating this side-channel threat, which causes *the leaks of sensitive user information out of* *the encrypted channel connecting a web application's client and server components.* As a first step, we focused on the leaks from packet sizes and sequences, not inter-packet timings.

There are two *necessary* conditions for such side-channel leaks to happen. First, sensitive data should "taint" network traffic: if they always stay within the client or the server side without any direct or indirect influence on the communication between these components, they are impossible to infer by the eavesdropper. Second, at least some privacy-critical properties of the data should be identifiable from the encrypted channel: tainting the network traffic by sensitive information does not necessarily imply that it can be inferred; the side-channel leaks happen only when packet sizes, sequences and other features of encrypted information flows vary in accordance with the content of the flows, and when such variations are consistent across different users and at different times.

Our prior research discovered that side-channel leaks from data flows exist in high-profile web applications. For example, consider the auto-suggestion list used in a personal health application; every keystroke it receives taints an AJAX request, which uses the letter entered to acquire from the server a list of suggestions with *distinctive* size, essentially enabling the eavesdropper to differentiate different keystrokes. Side-channel leaks from control flows are also realistic [13]. In a popular tax application we studied, one's family income affects the program's branch conditions. For example, depending on whether the income is below $115,000, above $145,000, or between $115,000 and $145,000, the application makes different decisions on the Student Loan Interest credit, which moves its execution into different sub-modules to handle the cases of "Full Credit", "No Credit" and "Partial Credit". The attributes of the web traffic associated with these sub-modules are distinguishable, which reveals the range of the user's income.

## 2.2 An Example

Sidebuster analyzes a web application's information flows to detect and quantify side-channel leaks. Here we use an example to explain how it works. Throughout the paper, we use "sensitive" and "private" interchangeably, and "taint" to refer to the situation that data has been directly or indirectly affected by the sensitive information labeled by the developer.



```
onChange() {
   if (UserType == "patient")
      loadPatientPage();
   else
      loadDoctorPage();
}
```

```
LoadPatientPage() { ... }
loadDoctorPage() { ... }
```

```
onChange() {
   temp = list1.getSelected();
   value1 = temp;
   rpc1(value1, callback1);
}
callback1(result) {
   list2.addItem(result);
}
```

```
rpc1(value1) {
   return
     getDecriptFromDB(value1);
}
```

```
onChange() {
   value2 = list2.getSelected();
   value1 = list1.getSelected();
   rpc2(value1, value2, callback2);
}
callback2 (result) {
   if (result) dialogBox("Succeed");
   else dialogBox("Failed");
}
```

```
rpc2(value1, value2) {
   return
     saveToDB(value1, value2);
}
```

**Figure 1: A simple example**

Figure 1 shows a simplified health information system. To use the system, one first needs to select a user type, either "patient" or "doctor". The selection posts an AJAX request to the server to download a program module to the browser. The functionality of the patient module is totally different from that of the doctor module. Thus the user type is essentially a branch condition that decides the application's execution paths.

The patient module consists of two list boxes as shown in the figure. They jointly enable the user to select an illness name: she is supposed to first choose the initial letter "A-Z" of the name using the first list box; this causes the browser to download `list2`, which contains the names of all diseases with that initial; the user's selection on that list sends a specific disease name to the server, which stores the item.

**Side-channel detection.** Our approach takes several steps to detect possible side channels in the application, as illustrated in Figure 2.

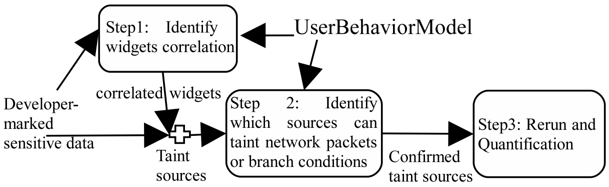

Step1: Identify widgets correlation
UserBehaviorModel
Developer-marked sensitive data
correlated widgets
Taint sources
Step 2: Identify which sources can taint network packets or branch conditions
Confirmed taint sources
Step3: Rerun and Quantification

**Figure 2: The working flow**

To perform an analysis, the developer of the web application needs to do two things. First, she must label in the source code the variables that carry private information, for example, `value2` that hold the name of the selected illness, as required by other information-flow analyzers such as JIF [36]. Second, the developer is supposed to identify a set of user-action sequences to be tested. This is because in a real-world web application, the number of possible action combinations, which correspond to different event sequences and orders for invoking different call-back functions, can be very large. Our analyzer therefore requires a pre-defined action specification as its input, which defines a set of possible user behaviors that developers want to analyze. In the example, one of the action sequences can be "clicking the patient radio button", "select in the first list", and then "select in the second list". The objective here is to determine whether these user actions leak the sensitive data to the eavesdropper; if so, the analyzer moves on to quantify such information leaks.

As stated earlier, one of the necessary conditions for the side-channel leaks is that sensitive data should taint network traffic. To identify such tainted network operations from web applications, Sidebuster performs two types of taint analyses: forward and backward. It is intuitive that once a piece of data is marked as sensitive, we want to examine whether it will eventually affect the network traffic. This is done through a standard forward taint analysis using user-labeled variables as taint sources (Step 2 in Figure 2). For this purpose, we implemented a taint analysis tool based on Soot [46], a Java code optimization framework that supports both data and control flow analyses. In the example, the variable `value2` directly taints `rpc2` within the call-back function of `list2`, which is identified by the forward taint analysis.

A prominent property of web applications is the extensive use of AJAX GUI widgets to assist the user in entering her input. In the example, even before `value2` has been delivered over the network, its content has already been partially revealed by one's selection of an item on `list1` and `UserType`. In order to tackle this situation, we need to identify (in Step 1 of Figure 2) all widgets related to the sensitive data, and include them as the taint sources for Step 2. This is achieved through a backward taint analysis: in Figure 1, Sidebuster treats `value2` as a taint target and analyzes all other widget inputs that directly or indirectly affect its content, which leads to the identification of `UserType` and API call `getSelected` as additional taint sources, and `rpc1` and loadPatientPage as other tainted network operations, in addition to rpc2. Note that here we also use a widget API as a taint source, as the widget's internal implementation is often opaque to application developers, who therefore can only label the API that returns the user's input. Also, the taint source UserType affects value2 through a control flow: it determines the download of the patient module, which leads to the generation of the private input held by value2. The content of this source can also be disclosed through the control flow, through the different traffic patterns when different modules are being downloaded from the web server. After the analysis, Sidebuster instruments all the tainted network operations for quantifying the amount of the information that can be inferred through these side channels. We elaborate the design and implementation of this detection technique in Section 3.

**Side-channel quantification.** The outcomes of the taint analysis could be inconclusive. We can only identify the program locations where sensitive information *could* be leaked out of an encrypted channel. Whether such leaks indeed happen and how serious they are can only be determined by looking at the relations between features of encrypted traffic and contents of taint sources. This is achieved in our research through a quantification analysis. To perform this analysis, the developer needs to specify the ranges of the values taint sources and targets can take. In the example, selection of user types has two choices, getSelected for list1 returns a letter in {‘A’,· · · , ‘Z’} and for list2 outputs 1 out of 260 different diseases, with 10 under every initial letter, for example. Sidebuster then automatically finds out the relations among those sources with regards to different tainted network operations. For example, rpc1 is tainted by list1.getSelected and can only be reached when the user type is “patient”, and rpc2 is tainted by UserType and getSelected from both lists. To quantify their joint side-channel leaks, Sidebuster strategically reruns the application from different instrumented call sites to understand how distinctive traffic patterns lead to the partitions of these value ranges. Specifically, we first check whether selection of different user types can be inferred from their traffic pattern. Then, Sidebuster analyzes rpc1 with different values from the range of list1.getSelected. Since rpc2 is found to be tainted by the inputs from both list1 and list2, it is analyzed with the values allowed to be taken by value2 under different initial letters returned from list1.

The outcomes of the analysis are combined and measured by loss of entropy, i.e., the number of bits of information revealed from these side channels. For example, if all 26 letters from list1.getSelected are found to be identifiable and all 10 disease names under each initial letter are broken into two equal-size sets by their traffic features, Sidebuster concludes that the amount of side channel leaks is 5.7 bits for the application. The details of such a quantification technique are presented in Section 4.

# 3. DETECTING SIDE-CHANNEL LEAKS

In this section, we elaborate how Sidebuster detects side channels within web applications. We first present the technique that identifies such vulnerabilities from a program's data and control flows, and then focus on AJAX-driven user interfaces to explicate how our approach addresses the special technical challenges they bring in. These techniques were implemented in our research to a prototype system for analyzing GWT-based web applications, which is also reported here.

## 3.1 Analyzing Program Information Flows

As stated in Section 2, side-channel leaks become possible when sensitive information is exchanged across the network, between a web application's client/server components. Therefore, the first step to detect this problem is to check whether the sensitive data are propagated to the network operations within the application and locate such operations when they are present[1]. This was achieved through an in-depth analysis of the program's data flows and control flows, which is also known as a taint analysis. For this purpose, Sidebuster includes an information-flow analyzer, which is built on a Java compiler infrastructure called *Soot* [46]. Soot takes Java source code or byte code as inputs and transforms them into Jimple code, an immediate representation (IR), which Sidebuster uses to perform a taint analysis. Our analysis extends the ForwardFlowAnalysis framework provided by Soot and is integrated into Soot as an additional transformation pass for taint analysis.

Before a taint analysis can be carried out, the web application developer needs to mark some “taint sources”, the variables that host sensitive information. Such taint sources are specified through an XML file sensitivedata.xml in our prototype. Starting from them, Sidebuster performs the analysis along both data flows and control flows in a web application, as described below.

**Leaks through data flow.** A data-flow leak happens when the content of a taint source affects the data to be sent through the network but does not change a web application's execution path. For example, in an auto-suggestion widget, different keystrokes to the input box result in different traffic features, but the application's control flow remains the same regardless of what the user enters.

To detect such leaks, we use the existing taint analysis capability of Soot. Here, the taint sources are the variables or API functions accommodating sensitive data and the taint sinks are the parameters of the APIs that trigger the communications between the client and server components. The taint analysis starts with a *UnitGraph*, a control flow graph (CFG) Soot creates for a Java program, with each node on the graph being a Jimple statement. Jimple has fifteen different types of statements but only eight of them appear in most GWT web applications[2]. For each statement, we defined a set of taint propagation rules, as presented in Table 1, which are used by Sidebuster to track the propagation of tainted data[3]. Our analyzer is also designed to support inter-procedural taint analysis: whenever a function invocation statement is encountered, Sidebuster locates the code of the function to build a new UnitGraph, and then uses its input parameters to determine the tainted variables within the function for the analysis on the new CFG.

**Table 1: Taint propagation rules**

| Statment | sample code | Rules |
|---|---|---|
| IdentityStmt | this := that | Taint(this) = taint(that) |
| AssignStmt | lhs = rhs1 OP rhs2 ; | Taint(lhs) = Taint(rhs1) OR Taint(rhs2) |
| InvokeStmt | f(a); | Taint(parameter0) = Taint(a) |
| ReturnStmt | a=f(b); | Taint(a) = Taint(c) where f(b) evaluates to c |
| ReturnVoidStmt | f(c) {return;} | (none) |
| IfStmt | if (a) goto label; | (none) |
| GotoStmt | goto label | (none) |
| SwitchStmt | switch(a) {case 1 ...} | (none) |

Like other taint analysis tools [37, 17], Sidebuster needs to address two standard technical hurdles, the presence of API functions and aliases. Analysis of API functions is known to be complicated and time consuming. Our design is meant to avoid getting into these functions whenever possible. Specifically, we took advantage of a command-line switch provided by Soot to exclude Java-library

[1] Note that tainting network operations does not necessarily indicate the existence of side-channel leaks, as the features of encrypted traffic may not reflect the content of sensitive user data. Whether such leaks indeed exist is actually determined by the quantification analysis, which also assesses the seriousness of the problem.

[2] Other statements, for example, EnterMonitor and ExitMonitor, are used by Java VM for some special purposes.

[3] Taint propagation from control flow to data flow is not supported in the prototype.

functions from our analysis. Instead, we built API models for some of these functions, which specify how tainted input parameters affect the outcomes of the calls. When such a model is not there for a function, our approach pessimistically taints all of its outputs once any of its input is tainted. The alias analysis, which is used to identify all the names for one tainted object, is made easy by the points-to analysis framework `SPARK` [32] offered by Soot. SPARK includes a method `reachingObjects` that returns the set of aliases for a given object or a variable.

Whenever a taint sink is found by Sidebuster, its related functions are instrumented for the follow-up quantification analysis. The taint sinks here can be parameters for Java library functions that cause network communications, for example, Socket, or user-defined functions that implement `RemoteService`, the Java interface implemented in GWT for any remote-procedure calls.

**Leaks through control flow.** Sensitive information can also be leaked out through control flows. Typically, such control-flow leaks happen when the application's execution path depends on sensitive user data, often a tainted branch condition, and when different paths exhibit distinctive traffic attributes. As a result, part of the sensitive data can be inferred by a network eavesdropper. For example, Figure 3 illustrates the part of the program logic within a tax preparing application for determining a user's eligibility for a tax credit: the branch conditions in the example are tainted by `income`, a variable containing the user's family income, which is specified by the developer as a taint source.

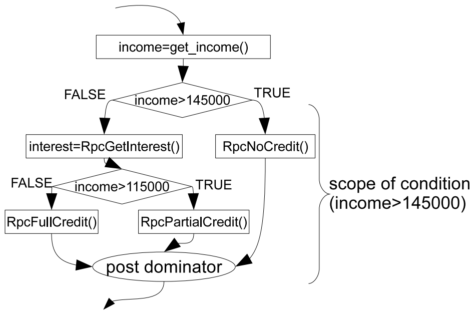


**Figure 3: Program logic of credit claiming**

Through the data-flow analysis, Sidebuster first finds out whether tainted information is propagated to a conditional branching statement. When this happens, a control-flow analysis ensues, which evaluates whether the content of the condition can be inferred by distinctive communication patterns associated with different branches of the statement. Specifically, our approach first seeks the *post dominator* of the condition, a program location at which all the branches converge, as illustrated in Figure 3. The post dominator is important because together with the conditional branching statement, it determines the *scope* of the control flow affected by the condition, which involves all the statements between these two program locations (see Figure 3). Within the scope, different branches are taken, leading to different program behaviors to be observed, according to whether the condition is TRUE or FALSE. The post dominator is located by Sidebuster using a Soot class `MHGPost DominatorsFinder`. Then, we extract from these branches their sequences of client-server interactions, which are typically in the form of remote procedure calls (RPC). These sequences are compared with each other to catch potential side-channel leaks: if different outcomes of a tainted condition lead to different RPC call sequences (in terms of the number of RPC calls on each sequence), Sidebuster reports a side channel vulnerability, because an eavesdropper can figure out which branch is taken (and thus the information about the branching condition) based on the observed communication caused by these calls. Otherwise, the analysis is inconclusive. In either case, we instrument all the call sites for a follow-up quantification study. This analysis will be performed recursively when tainted conditional branches are nested: that is, a tainted branch appears on the branch of another tainted condition.

For the example in Figure 3, the first condition is tainted by `income`. As a result, the analyzer examines all execution paths within its scope to extract the following RPC sequences: [RpcNoCredit] when `income`>$145000, [RpcGetInterest, RpcPartialCredit] when $115000 <`income`<$145000, and [RpcGetInterest, RpcFullCredit] when `in come`< $115000, for no credit, partial credit and full credit paths respectively. From these sequences, Sidebuster concludes that the condition "`income`>$145000" is disclosed, as an eavesdropper knows that there is only one round of communication (i.e., one RPC) when the condition is true, and two rounds (two RPCs) otherwise. The condition "`income`>$11500" is attached to two sequences with the identical number of calls, and thus needs to be further analyzed during the quantification stage.

**Cross client-server analysis**. During our analysis, the taint tag can go across the network, from an application's client component to its server component, and vice versa. The techniques for performing this cross client/server analysis depend on the programming languages used to develop the application. GWT applications use Java to implement both client and server components: the client-side code is ultimately converted into JavaScript, and the server runs a Java Serverlet.

In GWT, the standard client/server interactions are based upon remote procedure calls[4]. RPC is asynchronous: that is, a program makes such a call and then moves forwards without waiting until the outcome of the call returns; the processing of the return values is delegated to a call-back function registered during the call. When analyzing an RPC, our approach propagates taint information to the remote procedure and then tracks the information flow back to the client-side call-back function. The names of the procedure and the function can be easily extracted from the parameters of the RPC. Specifically, in GWT, the name of a remote procedure will be appended with a suffix string "Async" when it is invoked by the client. For example, for a procedure "RPCall()" on the server, the client needs to call it through the name "RPCallAsync()". Our analyzer removes the suffix from the parameter to an RPC before searching for the procedure from the server-side code.

## 3.2 Analyzing AJAX-driven User Interactions

AJAX-driven user interfaces, often known as GUI widgets, are widely used in web applications to help the user enter her input. For example, major search engines (Google/Yahoo/Bing) all provide autosuggestion widgets, which allow one to key in a couple of letters and then choose an item on a list to input the query word for a search. Such widgets pose special challenges for the taint analysis. First, even before a piece of sensitive input data are constructed by user-widget interactions, their content can already be disclosed by such interactions. In the above example, part of the query word can already be inferred by the operations for retrieving suggestion lists from the web server before the user finishes typing the whole word. Therefore we face the problem that given the complexity of widget-based user interactions, the relations between the user actions and the input eventually built are often indirect and less obvious to even the developers. For example, in Figure 1, though the content of `value2` is obviously generated by one's selection on `list2`, its connection with the operation on `list1` is less clear,

[4] GWT also supports other client/server interaction methods, for example, through URLs. However, RPC is the most commonly used approach and the one GWT recommends. For simplicity, our current implementation focuses on RPC but can be extended to accommodate other methods.

and even less so is its dependence on `UserType`. Second, widgets are often provided by the third party. Their source code can be unavailable to the developer. Even when the code is accessible, it is not practical to require the developer to understand the implementations and mark sensitive internal variables. We believe that only requiring the developer to label the input generated by a widget (e.g., query word) is much more realistic than manually identify the microscopic interactions (e.g. entering letters) with potentially multiple widgets related to that input. In our prototype, we developed a new technique that uses a *taint target*, i.e., the input formed by a widget, to *automatically* label the user actions related to its content, which is elaborated below.

**Backward taint analysis**. Our approach is to let the developer mark a specific taint target and then perform a backward taint analysis to find out all the widgets and their related user actions responsible for the content of the target. Here, the user actions are described by the API functions of these widgets that return the data that the user inputs. For example, GWT offers the autosuggestion functionality via a Java class called `SuggestBox`, which includes an API `event.getNativeKeyCode()` to return the letter that the user types. Such API functions will be tainted[5] by the backward analysis if the user actions are found to be related to the taint target. This analysis offers a convenient avenue for the developer to locate and evaluate all the widget interactions that can lead to side-channel leaks.

A natural way to perform such a backward taint analysis is to conduct a backward slicing [48] to identify all the program statements that affect the taint target. This can be achieved using standard slicing technologies [25]. Alternatively, we can first assume that every widget affects the taint target, and set all of its API functions that return user inputs as *hypothetical taint sources*. A forward taint analysis is performed on these sources to check whether information flows from these sources to the target. In the prototype of Sidebuster, we implemented the second approach.

**User behavior models.** An important issue in analyzing widgets and other event-driven programs is to determine the path of the execution triggered by user interactions. Typically, these programs include a set of call-back functions that hook different user input events, like keystrokes, mouse clicks and others. In response to different combination of user actions, different sequences of call-back function invocations happen, which leads to different execution paths a program follows and different outcomes of the taint analysis. For example, consider an autosuggestion list: whenever a user types a letter, the call-back function of the widget sends all the letters in its text box to the server. The problem here is that without knowing how the user enters letters, we have little idea how many letters, each of them carrying a distinctive taint label, taint the RPC of the function. Thus, analysis of such a program relies on a pre-determined set of user-event sequences.

To use our analysis tool, the application developer is supposed to provide a *user behavior model*, which specifies the user-event sequences to be evaluated. Such a model is included in an XML file. Following is an example:

```
<UserBehavior>
<Widget classes = "com.test.client"
          method = "onModuleLoad"
            name = "radio1"
          action ="onSelect"/>
<Widget classes = "com.test.client"
          method = "onModuleLoad"
            name = "list1"
          action ="onChange"/>
<Widget classes = "com.test.client"
          method = "onModuleLoad"
            name = "list2"
          action ="onChange"/>
</UserBehavior>
```

The behavior model defines the event sequences to be analyzed. Each sequence consists of a set of nodes, with each of them containing the information of the widget and its specific event. In the example, `classes`, `method` and `name` are used to jointly identify a widget within an application, and `action` indicates the user event: specifically, `onChange` is an event that happens with the update of the user's select on the list. A challenge here is to find out the call-back functions of individual events, which are registered during the application's runtime. Sidebuster locates such functions from event registration methods such as `addOnChangeHandler()` during a static taint analysis.

**Hidden control flows.** During an interaction with a web application, a user's behavior, e.g., selection of an item and click on a certain link, often carries sensitive information. Though in many cases, such information can be described by certain program objects like a tainted variable, there are situations where no explicit taint source or target exists to accommodate the information. For example, consider a user who is expected to push one of two buttons, each triggering a different call-back function. Her action here, i.e., choice of the buttons, is confidential. However, no single variable and function inside the program actually hosts this piece of sensitive information: it is just described by different execution paths being triggered.

We found in our research that such hidden control flows exist in real-world web applications. To detect their side-channel leaks, Sidebuster allows the developer to group a set of user events and mark them with the same taint label. When such grouped events are specified, our approach treats their call-back functions as different branches of a tainted path condition, and performs a control-flow taint analysis on them as described in Section 3.1.

# 4. QUANTIFYING SIDE-CHANNEL LEAKS

After side channel vulnerabilities have been identified from web applications, we need to understand how serious the problem is and whether it deserves mitigation efforts. To this end, we also propose a suite of techniques for quantifying the information leaked from these channels. Our techniques are based upon a black-box test guided by the information about the side channels detected by the taint analysis and the relations among different taint sources. More specifically, our approach is to rerun an application to collect the attributes of the traffic associated with different contents of taint sources. Then, we evaluate to what extent such attributes help an eavesdropper partition the value ranges of those sources. These two steps are elaborated in the rest of the section.

## 4.1 Rerun Testing

The idea of rerun testing is to continuously rerun portions of a web application's code involving side channels, as reported by the detection stage, to understand how the traffic attributes generated thereby classify different values taint sources can take. In our research, we implemented such a testing tool using `HtmlUnit` [3], an open-source web testing tool. The web traffic created during a test is recorded by `Jpcap` [5], a Java library for packet capture.

**Rerun guidance.** A rerun test is guided by the vulnerability information acquired from the taint analysis. Most importantly, each test run should start from the program location where the content of the related taint source can be legitimately adjusted and always

[5] For a tainted function, each time it is called, the result it returns carries a different taint label.

follows the execution path that leads to the side channel under the test. This can be achieved through symbolic execution [29], a static analysis that uses symbols, instead of real values, as inputs to a program to analyze its execution. Performing symbolic execution on the taint-propagation path that leads to a side channel, we can get its *path condition* and can therefore evaluate the values of the taint source that satisfies the condition. Alternatively, we can take advantage of the test cases that the developer uses for evaluating the functionality of the application. Since side channels are often attached to important client/server interactions, the test cases leading an execution to such locations often exist and can thus be reused.

For GUI widgets, the rerun guidance also needs to include other information. Specifically, Sidebuster has a guidance file "rerun.xml" that is generated in the detection phase and contains the following information for a widget: (1) `name` for locating it on a web page, (2) `type` for extracting legitimate values for testing, which we explain later, and (3) `user actions` for triggering the event sequences to be evaluated, which come from the user behavior model.

The rerun test performed on widgets also utilizes another piece of guidance information – the relations among tainted user inputs[6]. Knowledge of such relations is important because it can significantly improve the performance of a rerun test. For example, consider two AJAX lists with 10 elements each; if the user's selections on these lists are independent from each other, we can evaluate them separately, which reruns the widget 20 times; on the other hand, if the selections are correlated, then we have to perform the test for 100 times. Such relations are automatically discovered by Sidebuster during the taint analysis: if a network operation is found to be tainted by two sources, these sources are considered to be connected. For example, the letters typed into `SuggestBox` are related, as they are pieced together to retrieve a suggestion list from the server. Using the relations, Sidebuster maintains a taint-source vector $V = \langle T_1, \cdots, T_n \rangle$, where $T_{i \in \{1, \cdots, n\}}$ is a set of correlated sources, and the sources in two different sets are independent from each other. The space of $V$ can be partitioned by the side-channel leaks from individual taint sources. The objective of the quantification analysis is to determine the total amount of information disclosed from such a partition.

**Other test preparations.** Before testing, another important issue we need to address is to determine the range of the values that each taint source can take. Some of such ranges are given by the developer, while others can be automatically identified. Particularly, those associated with widget elements, such as check boxes and list boxes, can be analyzed to find out their legitimate contents, either directly from their properties (e.g., check boxes), or from their hosting HTML pages (e.g., List boxes). For the elements like text box and variables, Sidebuster lets the developer specify their value ranges. For example, one has to indicate that the user's input letter (i.e., return value of `event.getNativeKeyCode()`) will be one of the English alphabets.

As mentioned before, the client-side component of a GWT application will be converted into JavaScript code. The problem here is that the name of the widget to be tested changes during this transformation. This can typically be resolved by looking at the combination of a widget's type and container information. Another approach is to instrument the Java code of the application with GWT API `getElement().setId()`, which forcefully sets the ID of a widget to a pre-determined one.

**Rerun.** During a rerun test, Sidebuster executes a web application repeatedly with different taint-source vector $V$. The web traffic generated during such executions is recorded by `Jpcap`, which is invoked by our instrumentations. The user inputs during the test are automatically generated by our analyzer. After one test, the application is rolled back to where individual taint sources in $V$ are specified, using `HtmlUnit`'s checkpoint/rollback mechanism. These sources are then updated for the next round.

Since taint-source elements in $V$ are independent from each other, we can evaluate them one by one. In the case that an element contains only one taint source, we try to enumerate all its legitimate values and record the traffic attributes corresponding to these values if possible. If the source's value range is continuous or too large, we allow the options to either randomly sample some values from the range, or partition it into segments to randomly pick up a value from each of them.

For $T_i \in V$ that includes multiple taint sources, a re-run test needs to be more strategic. These sources are typically the user's inputs to a set of related widgets. It is conceivable that all their value combinations need to be checked. This, however, turns out to be unnecessary in practice. For example, consider an auto-suggestion widget with an event sequence in which five letters are consecutively entered. Though the possible combinations of these letters can be as large as $26^5$, only a few thousand of them actually bring in non-empty suggestion lists. Sidebuster has been designed in a way that it organizes all sources in $T_i$ into a tree: if a source receives inputs or is defined earlier than another source, the former becomes the ancestor node of the latter on the tree. In the autosuggestion example, we put the 26 possible letters for the first keystroke on the first layer, just beneath the root; each letter parents other 26 letters for the second keystroke and so on. During the test, Sidebuster performs a breadth-first traversal of the tree: it first evaluates all the inputs on the first level and then goes down to the second level and so on. Important to this process is that once the values on a branch until a certain level are found to be no longer valid (e.g., no suggestion list for a certain sequence with three letters), Sidebuster stops testing all the nodes on the next level attached to that branch. In the end, all the leaves of the tree describes the space of $T_i$.

**Discussion.** A challenge for the rerun test is possible large number of test cases for each $T_i$, which makes an exhaustive evaluation unrealistic. As mentioned before, we can always control the scale of the test through random sampling. On the other hand, our prior research [13] shows that a prominent property of web applications is low entropy inputs: even a single keystroke or a mouse click could trigger AJAX interactions and produce web traffic. This is a fundamental cause of the side channel problem. It also makes the size of the test-case space manageable in many practical situations.

Another problem is that the attributes of the traffic associated with the same user input can be different between users. This happens when the traffic also carries some user-specific information, such as cookies, as discovered in our prior research. To find out what exactly the adversary can learn from encrypted web traffic, Sidebuster also performs a cross-user test to extract *invariants* from the traffic attributes related to a specific input. For example, if we find that among all the packets generated when a list item is selected, only one has an invariable size from user to user, we can pick up that packet as the *signature* for the selection. Using traffic signatures to determine confidential user data offers a more realistic assessment of the side-channel leaks.

## 4.2 Quantification

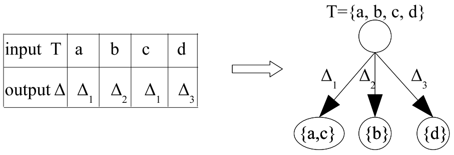
After the rerun test, Sidebuster quantifies the amount of the information revealed from detected side channels. Denote the space of $V$ by $\Omega$. What we intend to measure here is the loss of entropy after the space has been partitioned by the side-channel information into disjoint sets so that an eavesdropper can tell the sensitive user

[6] Tainted user inputs refer to those whose related widget API functions are tainted. For example, the letter one enters into a `SuggestBox` is tainted if the API `event.getNativeKeyCode()` is tainted by the backward taint analysis.

data in different sets apart. In our research, we adopted *conditional entropy* as the measurement, a concept that has also been used in prior research on quantitative information flow analysis [35].

Let $\Delta$ be a set of traffic attributes, which in our research describes a sequence of directional packet lengths [13]: for example, $(64 \Rightarrow, \Leftarrow 1024)$ represents a 64-byte request from browser and a 1024-byte response from web server. Before they are observed by the adversary, the entropy of sensitive user data is $H(\Omega) = \log_2(|\Omega|)$. After that, it becomes $H(\Omega|\Delta) = \sum p(\Delta_i)H(\Omega|\Delta_i)$, where $p(\Delta_i)$ is the probability for the attributes $\Delta_i$ to appear, and $H(\Omega|\Delta_i)$ is the conditional entropy given the observation of $\Delta_i$. Since every element on a vector $V$ is independent from others, we can calculate the conditional entropy $H(\Omega|\Delta)$ by adding the entropies of individual elements $H(T_i|\Delta)$ together. This property allows us to focus on the entropy of each element. In the following, we show how to quantify the side-channel leaks from the element, when it contains a single taint source or multiple sources.

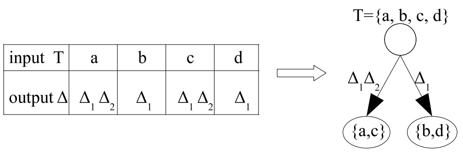
**Quantifying the leaks from a single taint source.** We first consider simple cases in which information leaks from a single source through data flows or control flows. Here we overload the symbol a little bit, using $T$ to represent the set of all possible values the source can take. To calculate $H(T|\Delta)$, we first group these values into disjoint sets according to their distinctive traffic attributes (e.g., different packet sizes), as observed in the rerun test, and then compute the conditional entropy for each attribute (e.g., each size), that is, the entropy of its set. These entropies, weighted by the probabilities of taking the values in individual sets, are summed up to the conditional entropy of $T$ with regards to those attributes. For example, Figure 4 shows a taint variable with four distinctive values T ={a, b, c, d} and three different traffic attributes (i.e, three different packet sizes/sequences). These attributes, denoted by $\Delta_1$, $\Delta_2$ and $\Delta_3$, classifies $T$ into the partition: { $\Delta_1 \rightarrow \{a, c\}$, $\Delta_2 \rightarrow \{b\}$, $\Delta_3 \rightarrow \{d\}$}. Assuming that each value has the same chance to be taken by the variable[7], we have $p(\Delta_1) = 0.5$ and $p(\Delta_2) = p(\Delta_3) = 0.25$. $H(T|\Delta_1) = H(\{a, c\}|\Delta_1) = 1$ bit, $H(T|\Delta_2) = H(\{b\}|\Delta_2) = 0$ bit, $H(T|\Delta_3) = H(\{d\}|\Delta_3) = 0$ bit, Therefore, the conditional entropy of $T$ becomes 0.5 bits. This indicates that with the observation of the side-channel information (i.e., the attributes), the entropy of the variable goes down from 2 bits to merely 0.5 bit.



| input T | a | b | c | d |
|---|---|---|---|---|
| output Δ | $\Delta_1$ | $\Delta_2$ | $\Delta_1$ | $\Delta_3$ |

**Figure 4: Quantify the information leaks through data flow**

The second example in Figure 5 shows the information leaks from control flows. Again, a tainted variable has a value range {a, b, c, d}. A branch condition tainted by the variable becomes TRUE, when the variable takes $a$ or $c$, and FALSE otherwise. The TRUE branch is characterized by two traffic attributes $\Delta_1$ and $\Delta_2$, while the FALSE branch has only one attribute $\Delta_1$. Given a uniform distribution over the range, we have $p(\Delta_1\Delta_2) = 0.5$ and $p(\Delta_1) = 0.5$, and $H(T|\Delta_1\Delta_2) = H(\{a, c\}) = 1$, $H(T|\Delta_1) = H(\{b, d\}) = 1$. Thus, we conclude that the observation of these attributes reduces the entropy of the variable from 2 bits to 1 bit.

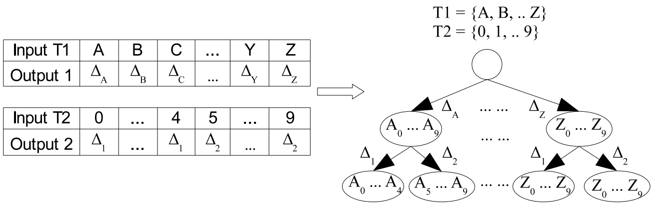


| input T | a | b | c | d |
|---|---|---|---|---|
| output Δ | $\Delta_1\Delta_2$ | $\Delta_1$ | $\Delta_1\Delta_2$ | $\Delta_1$ |

**Figure 5: Quantify the information leaks through control flow**

**Quantifying the leaks from multiple sources.** Information leaks from multiple related taint sources need to be quantified over the tree for the rerun test. As described before, the leaves of the tree represent all the legitimate value combinations of those sources. Here, we again use $T$ to represent the set of these combinations. The amount of information disclosed from this set can be quantified in the same way as the case of a single taint source. Figure 6 shows how to quantify the information leak for the example introduced in Figure 1.

This example has two user inputs: $T_1$ is the set of the initial letters of the disease names on `list1` and $T_2$ is the set of the item indices of the disease names on `list2`. Suppose all 26 letters from $T_1$ generate different traffic attributes $\Delta_A$ to $\Delta_Z$, and all 10 disease names under each initial letter have only two identifiable traffic attributes: the first five items generates $\Delta_1$ and the others produces $\Delta_2$. Since these two sources are correlated, their traffic attributes divide the leaves of the tree illustrated in Figure 6. Assuming that each disease name is selected with an equal chance, we have: $p(\Delta_A\Delta_1) = p(\Delta_A\Delta_2) = ... = p(\Delta_Z\Delta_2) = 1/52$ $H(T|\Delta_A\Delta_1) = H(T|\Delta_A\Delta_2) = ... = H(T|\Delta_Z\Delta_2) = \log_2(5)$. As a result, the total entropy is reduced from 8.02 to 2.32 bits.



| Input T1 | A | B | C | ... | Y | Z |
|---|---|---|---|---|---|---|
| Output 1 | $\Delta_A$ | $\Delta_B$ | $\Delta_C$ | ... | $\Delta_Y$ | $\Delta_Z$ |

| Input T2 | 0 | ... | 4 | 5 | ... | 9 |
|---|---|---|---|---|---|---|
| Output 2 | $\Delta_1$ | ... | $\Delta_1$ | $\Delta_2$ | ... | $\Delta_2$ |

**Figure 6: A quantification example with multiple correlated user inputs**

# 5. EVALUATION

## 5.1 Experiment Settings

To evaluate Sidebuster, we ran our prototype on 6 applications, whose descriptions are presented in Table 2, to study the situation when an eavesdropping adversary tries to use packet sizes and sequences of the HTTPS traffic produced by these applications' client/server interactions to infer their user data. Among the applications, the first three were synthesized to simulate the functionalities of the high-profile commercial applications studied in our prior research [13], including a leading health information system, one of the most popular tax preparation applications and popular search engines such as Google/Yahoo/Bing. We do not name some of those applications here, per requests from related organizations. In absence of their source code, we built part of their user-interaction modules over GWT, using the data collected from the real applications. Specifically, the first program, which supports a query suggestion widget adopted by most search engines, contains the suggestion data we dumped from Bing. The second one has a user interface that includes the disease names extracted from the health information system. The third application involves the program logic of tax software. The other three web applications are open-source, which we downloaded from the Internet. We used

[7] We also used this uniform distribution assumption in our evaluation. The developer can adopt another distribution if it is known.

both synthesized and real applications in our research due to limited access to the source code of real-world applications: though GWT has been widely used to develop web applications, most of them are actually closed source.

Each of these applications has distinctive side channels. The query suggestion program discloses the query words a user enters through the letters she types. The health profile program includes multiple widgets whose interactions leak information. The tax program has side-channels in its control flows. The three real applications expose user behaviors and data through the traffic generated by button clicks or link selections.

The experiments on static analysis were conducted on a ThinkPad X61 laptop, with 3GB memory and a Core 2 Duo 2.1GHz CPU. The quantification analysis utilized a Apache Tomcat 6.0 web server running on a RedHat Enterprise Linux 5.0 machine and a FireFox 3.6 browser on a Windows Vista client. The server is equipped with a Core 2 2.66GHz CPU and 4GB memory. The client has a AMD Turion X2 Dual-Core 2.00GHz CPU and 3GB memory.

## 5.2 Detection

To detect side channels, we first labeled variables in individual applications. For each application, our prototype first performed a backward taint analysis to identify the user inputs related to the contents of the variables and then a forward analysis to find out all the tainted network operations. We also built a user-behavior model for each application, as an input to our static taint analysis. The experiment results are summarized in Table 3.

**Data-flow side channels.** Sidebuster reported that Application 1, 2, 4 and 5 had data-flow side channels: the parameters of some of their remote procedure calls were tainted in our analysis. More specifically, the `SuggestBox` widget was found to send individual input letters to retrieve data from a web server. In a similar way, *GWT Advanced Table* used `filterText`, which was specified by our user model to contain two letters, to gather the data including the string from the server side. *DynaTable* had a tainted variable `startrow` that indicated to the server the course page a student was looking at. The variable propagated its taint to the RPC for such interactions. The leaks in the health profile came from the user's clicks on the first letters of certain disease names and the selections of the names afterwards. Such leaks, like the problem in `SuggestBox`, were identified through the backward taint analysis: in this example, the tainted variable hosting disease names was found to be related to the RPC `sendNameToServer` that delivered the initial letters and the selected items, and updated the contents of the list of disease names.

**Control-flow side channels.** Our prototype also detected the control-flow side channels in the *Tax credits* and *ETE2009 Demo* applications. The tainted variable in the Tax program, `agi`, was found in the branch conditions that led to the execution paths with different numbers of RPC functions. As a result, the branch taken by the program and relatd `agi` values can be easily determined by an eavesdropper. ETE2009 Demo gives an example of hidden control flows: it allows the users to click on different links, each leading to a different call-back function. Sidebuster discovered that invocations of these functions produced different number of RPC requests and therefore revealed the user's action.

We manually checked the problems reported by our prototype, and found that the detection did not cause any false positives. Also, all the known side-channel problems in Application 1, 2 and 3, as described in [13], were detected.

## 5.3 Quantification

We further measured the information leaks from those applications, using our quantification tool. All the user inputs in our experiments were automatically produced by `HtmlUnit`, in accordance with the user models. During the experiment, Sidebuster recorded the sizes and the sequences of the packets observed from the instrumented RPC call sites. Such information was used to calculate the conditional entropies of taint sources.

**Leaks from individual applications.** Whenever a letter was typed into auto-suggestion application, its `SuggestBox` automatically passed the query to Bing to acquire a suggestion list. This allowed us to estimate the information leaks through the same functionality used in real-world search engines. In our research, we tested all the combinations with two letters. The entropy of the query word before the test was $\log_2(26^2) = 9.4$ bits. After the test, 0 bit was left. This indicates that the adversary can unambiguously determine the query words from packet sizes and sequences. The health profile program utilized the data collected from a leading online health information application, which was also studied in our prior research [13]. There are totally 2508 disease conditions included in the data, which amounts to 11.29 bits of information. After the rerun test, we found that the side-channel leaks reduced the entropy to merely 2.38 bits. The tax program was designed to test one's eligibility for various tax credits. It included thirty different levels of the adjusted gross income (AGI) from 0 to $150,000 with a step of $5,000. The entropy here is 4.91 bits. After the test, it dropped to 1.3 bits.

The three real-world applications had their own data-sets. Although such data are not extremely confidential, the way in which they are exchanged in the applications bears a strong resemblance to those involving high sensitive user data, as we observed in our prior research [13]. Therefore, evaluation over these applications still provides useful information about the efficacy of our techniques. Specifically, DynaTable came with 100 course entries, organized into pages with each page containing 15 courses. Our analysis reduced the entropy of the pages being viewed to 0, which amounts to reducing the entropy of the courses from $\log_2(100)$ bits to $\log_2(15)$ bits. The GWT advanced table leaked information when it sends the user's input letters to the server. Different from `SuggestBox`, the application did not generate traffic for every letter. Nevertheless, its distinctive traffic attributes could still enable a side-channel attack: for example, the adversary could build a "dictionary" to record the mappings from traffic attributes to the content of input strings; when the number of legitimate strings (those with non-empty responses from the server) is small, such an attack becomes realistic. In our experiment, we found that such a traffic analysis could reduce the entropy of a two-letter string from 9.4 bits to 0 bits. The control-flow leaks from ETE2009 Demo were also found to be serious. With 7 different options, the entropy of the user's selection was 2.8 bits. Our analysis shows that almost all such information was given away through the side channel: only 0.68 bits were left after our test.

## 5.4 Performance

We also measured the performance of Sidebuster, i.e., the time spent on detection and quantification of side-channel leaks, as reported in Table 3. This study shows that the overheads of Sidebuster are moderate: the taint analysis on source code was typically accomplished in two to four seconds; time spent on rerun differed from case to case, depending on the data used in an application. For all applications tested in our study, both analysis steps were completed in about 30 minutes.

# 6. DISCUSSION

The prototype we have implemented only works on GWT-based applications. We chose GWT because it is one of the most widely used platforms for developing web applications, and because it is

**Table 2:** Experiment applications

| App | name | Description |
|---|---|---|
| 1 | Suggestion Box | It simulates the functionality of auto-suggestion widgets widely used in web applications such as web search engine. Whenever the user types a character, it will pop up a list of input entries the user can choose from. The testing data used in the program came from `bing.com`. |
| 2 | Health information system | It simulates the functionality of a real web application that manages users' health profiles. It has two inputs: the first one asks one to choose an initial letter of her/his health condition, and the second one lets user select from a list of conditions with that initial letter. The data also came from a real online health information application. |
| 3 | Tax credits | It describes the program logic of Tax preparing web applications. The program first asks user to choose her/his AGI (which was discretized with a $5,000 interval). Then it tries to determine whether the user is eligible for certain tax credits like student loan interest credit. |
| 4 | DynaTable | This is an application that comes with GWT SDK [18], which shows a school schedule for professors and students. Such information is organized into pages and the user can click on buttons to walk through these pages. |
| 5 | GWT advanced table | The application came from Google Code [2]. It demonstrates the usage of *GWT Advanced Table*, a complex widget that illustrates the contents of tables retrieved from a database. The application has a filter function that only shows the records containing a specific string. |
| 6 | ETE2009 Demo | This is an open-source program [4] with multiple links that trigger different call-back functions. |

**Table 3:** Effectiveness and Performance

| App | Types of Side-Channel Leaks | Detected | Entropy before test (bits) | Entropy after test (bits) | Detection Time (seconds) | Rerun Time (minutes) |
|---|---|---|---|---|---|---|
| 1 | Data flow leak: user input letters taint one RPC parameter. | Yes | 9.4 | 0 | 2 | 12 |
| 2 | Data flow leak: disease names taint one RPC parameter. | Yes | 11.29 | 2.38 | 3 | 33 |
| 3 | Control flow leak: user incomes lead to different execution paths. | Yes | 4.91 | 1.3 | 2 | <1 |
| 4 | Data flow leak: the start row of each page taints one RPC parameter. | Yes | 2.58 | 0 | 4 | <1 |
| 5 | Data flow leak: the data filter string taints one RPC parameter. | Yes | 9.4 | 0 | 3 | 12 |
| 6 | Leak through a hidden control flow: different user actions trigger differnt RPC sequences. | Yes | 2.8 | 0.68 | 4 | <1 |

based upon Java, whose analysis tools (e.g., soot, SPARK, Jpcap) are publicly available. GWT is not the only dominant web application platform. For example, ASP.Net and PHP are also widely used in the industry. However, we believe that it is completely possible to build development tools for these platforms based upon the fundamental idea of our approach, i.e., information-flow based side-channel detection and quantification analysis.

Our current design does not track the taint propagation from the control flow to the data flow because this would otherwise result in too many variables being tainted. This is a well-known hurdle in automatic program analysis, and techniques exist to mitigate such a "taint-explosion" problem [41].

During the rerun testing, we assume that user actions do not cause server-side writes (e.g., modifications or deletions in the back-end database). This is because our current implementation only rolls back the states in the browser. If some user actions indeed cause server-side changes, the rerun testing will require preserving the server-side states. Note that for many web applications, their server states are maintained in the back-end databases. Most database systems have checkpoints and roll-back functionalities, which can be incorporated into our rerun tool to handle most server-side modifications.

We rely on the developers to specify the user action model, which could increase their workloads and also result in inaccurate models. As discussed before, such a model is important to the side-channel analysis because web applications are event-driven: the sequence of events often determines the path of execution, which can be hard to predict without understanding the user's behavior. It should be noted that the behavior model may not be derived from the source code - for the same piece of source code, the user often has many possible action sequences. A possible avenue to facilitate automatic generation of more accurate model can be analyzing the user's interactions with a similar application to identify common actions, which will be studied in our follow-up research.

Like other software testing, completeness of a side-channel analysis is such a lofty goal that it is often unachievable in practice. The problem is further complicated by the fact that many side-channel leaks are actually related to the domain knowledge of a web application [13]. Further research is needed to incorporate such information into a detection mechanism.

# 7. RELATED WORK

**Side-channel leaks.** Side-channel leaks have long been known: some documented attacks even date back to World War I [21]. More recent studies on side-channels of encrypted communications have been conducted in various contexts, including secure shell [42, 50], voice over IP [49], multimedia data streaming [40], and web traffic [14]. Specifically about the web, it is well recognized that the anonymity of web browsing is very difficult to ensure (despite the use of anonymity channels such as Tor [39], Mix [24]) because webpages can often be fingerprinted by their side-channel characteristics [45, 12, 19]. Also, as we summarized in the introduction section, our recent work [13] demonstrated that side-channel leaks become a very serious problem for web applications. This work directly motivates our research on Sidebuster.

**Web application vulnerabilities and testing tools.** The study of web application vulnerabilities is a very broad area, in which well-known vulnerabilities include cross site scripting [1], SQL injection [6], and others. The vulnerabilities discussed in this paper are in the category of information leakage. To detect the vulnerabilities in a web application, a number of testing tools have been proposed. These tools, depending on whether the source code access is required, can be white-box or black-box. Our technique falls into the category of white-box testing, which typically uses information flow analysis and model checking techniques to uncover web vulnerabilities. For example, Lam et al. [31] defines a language with which users can declare information flow patterns, and detect the SQL injection and cross site scripting vulnerabilities whose patterns deviate from the specified ones. Huang et al. [26] proposes to first statically check whether a code is vulnerable to injection attacks, and insert a runtime guard to secure the program without user intervention. There are also various other papers of this kind [47, 28, 27]. Our approach, though also using information flow analysis, is distinct from the previous work that detects information leaks by simply looking at whether tainted data is propagated to the network. What we care about here is more complicated: for example, whether different contents of the tainted data will trigger different traffic patterns observable to an eavesdropper. The new challenge asks for significant additions to the existing analysis techniques, including the RPC sequence analysis, the rerun testing and the quantification of information leaks.

**Quantitative information-flow security.** Quantitative information-flow analysis was first proposed by Denning [20] in 1982, in which the data in a program are associated with a hierarchy of privilege levels, and are protected according to the non-interference principle [23] (low-security variables should not be affected by high-security data). A recent survey on this line of research is given by Mu [34]. Some of the prior approaches [10, 15, 33] statically analyze the source code of a program to measure the information disclosed from its outputs. Different from the programs studied in those approaches, web applications depends on their underlying web platform (web server and browser) to run. Without direct access to the source code of the platform, we have to resort to a combination of static and dynamic analysis, using rerun tests to quantify information leaks from observable traffic features, particularly packet lengths and sequences.

Effort has also been made to estimate the quantity of the information leaked from sensitive data given the relations between the content of the data and observations (e.g., timings) [9, 30, 43, 16]. For example, Clarkson et al. [16] proposes a model to measure the accuracy of an adversary's beliefs that are updated by his observations of a program's execution; Boris et al. [30] quantifies the information leaks from the secret keys of cryptographic systems through combining attack strategies with information-theoretic metrics; Standaert et al. [43] proposes a framework for analyzing whether a cryptographic implementation is secure by quantifying the effect of practically relevant leakage functions.

At a high level, our approach follows the similar idea, that is, quantifying information leaks based upon the relations between sensitive data and the adversary's observations. The conditional entropy metric we adopted is also standard [9]. However, prior approaches cannot be directly applied to determine such relations from real-world web applications: a web application often involves complicated, stateful user interactions and generates web traffic indirectly through its underlying platform, making establishment of the data-observation relations nontrivial. Sidebuster has been designed to deal with state transitions and user interactions within these applications, based upon a suite of new techniques including guided re-runs, use of user behavior models and analysis of aggregate information leaks from multiple user inputs. Also missed in the prior research is a quantification study of the information disclosed by packet lengths and sequences, which our approach focuses on.

Recently Borders and Prakash [11] proposed a method to quantify leaks in network traffic. The target scenario is not the side-channel leaks, but a malicious sender smuggling stolen data in the network traffic trying to avoid detection. They proposed an approach to quantify the bandwidth available to the malicious sender. What we target in this paper is quantification of entropy losses of sensitive data in benign applications to network eavesdroppers.

# 8. CONCLUSION

With the extensive use of web applications, side-channel information leaks, which are inherent to those applications, are becoming a non-negligible threat to private user data. Recent study shows that people's health information, family incomes, investment secrets and other sensitive data are being leaked by a set of high-profile, top-of-the-line web applications through their side channels [13]. Mitigation of such a threat is found to be highly non-trivial, requiring new methodologies to be developed to find the side-channel vulnerabilities in web applications and fix them. In response to this urgent call, we propose in this paper Sidebuster, the first automatic tool for detecting and quantifying side-channel leaks in web application development. Our techniques first statically analyze a web application's information flows to detect its client/server interactions tainted by sensitive user data, and then perform a dynamic analysis on related network operations to determine the amount of information being disclosed. Sidebuster can effectively handle special features of web applications, particularly use of AJAX GUI widgets. Our implementation of the approach, which works on GWT, has been demonstrated to be effective at determining the existence and seriousness of the side-channel problems in real-world applications.

# Acknowledgements

We thank Stephen Chong, our shepherd, for the guidance on preparing the final version of this paper, and anonymous reviewers for their insightful comments. We also thank Emre Kiciman at the MSR for valuable discussions. This work was supported in part by NSF Grants `CNS-0716292` and `CNS-1017782`, and AFRL `FA8650-09-D-1639`. Rui Wang was also supported in part by the Microsoft Internship program.

# 9. REFERENCES

[1] The cross-site scripting (xss) faq. `http://www.cgisecurity.com/xss-faq.html`.

[2] gwt-advanced-table. `http://code.google.com/p/gwt-advanced-table/`.

[3] Htmlunit - welcome to htmlunit. `hhtmlunit.sourceforge.net/`.

[4] Introduction to gwt – ete 2009. `http://gwtsandbox.com/ete2009`.

[5] jpcap – a network packet capture library for applications written in java. `http://jpcap.sourceforge.net/`.

[6] Sql injection. `http://en.wikipedia.org/wiki/Sql_injection`.

[7] Http secure. `http://en.wikipedia.org/wiki/HTTP_Secure`, 2010.

[8] Wi-Fi Alliance. Wi-fi protected access: Strong, standards-based, interoperable security for today's wi-fi networks. White paper, University of Cape Town, April 2003.

[9] Michael Backes and Boris Köpf. Formally bounding the side-channel leakage in unknown-message attacks. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 517–532, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, Washington, DC, USA, 2009. IEEE Computer Society.

[11] Kevin Borders and Atul Prakash. Quantifying information leaks in outbound web traffic. In *IEEE Symposium on Security and Privacy*, pages 129–140. IEEE Computer Society, 2009.

[12] David Wagner Bruce, David Wagner, and Bruce Schneier. Analysis of the ssl 3.0 protocol. In *Proceedings of the Second UNIX Workshop on Electronic Commerce*, pages 29–40, 1996.

[13] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *The 31st IEEE Symposium on Security and Privacy*, pages 191–206, May 2010.

[14] Heyning Cheng and Ron Avnur. Traffic analysis of ssl encrypted web browsing. `http://www.cs.berkeley.edu/~daw/teaching/cs261-f98/projects/final-reports/ronathan-heyning.ps`, 1998.

[15] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal Computer Security*, 15(3):321–371, 2007.

[16] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, 2009.

[17] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206, 2007.

[18] Google Code. `http://code.google.com/webtoolkit/`, as of 2009.

[19] George Danezis. Traffic analysis of the http protocol over tls. `http://research.microsoft.com/en-us/um/people/gdane/papers/tlsanon.pdf`, as of June 2010.

[20] Dorothy Elizabeth Robling Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[21] Jeffrey Friedman. Tempest: A signal problem. *NSA Cryptologic Spectrum*, 1972.

[22] Jesse James Garrett. Ajax: A new approach to web applications. http://adaptivepath.com/ideas/essays/archives/000385.php, February 2005.

[23] Joseph Goguen and Jose Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.

[24] Philippe Golle, Xiaofeng Wang, Markus Jakobsson, and Alex Tsow. Deterring voluntary trace disclosure in re-encryption mix networks. In *Proc. of the 2006 IEEE Symposium on Security and Privacy*, pages 121–131. IEEE Computer Society, 2006.

[25] Mark Harman and Robert M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, Jan 2001.

[26] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[27] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258–263, Washington, DC, USA, 2006. IEEE Computer Society.

[28] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *ICSE'09: Proceedings of 31st IEEE International Conference on Software Engineering*, pages 199–209, Washington, DC, USA, 2009. IEEE Computer Society.

[29] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[30] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296, New York, NY, USA, 2007. ACM.

[31] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 3–12, New York, NY, USA, 2008. ACM.

[32] Ondřej Lhoták. Spark: A flexible points-to analysis framework for java. `http://plg.uwaterloo.ca/~olhotak/pubs/thesis-olhotak-msc.ps`, 2002.

[33] Pasquale Malacaria. Assessing security threats of looping constructs. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 225–235, 2007.

[34] Chunyan Mu. Quantitative information flow for security: a survey. Technical Report TR-08-06, Department of Computer Science, King's College London, September 2008.

[35] Chunyan Mu and David Clark. Quantitative analysis of secure information flow via probabilistic semantics. In *International Conference on Availability, Reliability and Security*, pages 49–57, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[36] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[37] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, 2005.

[38] Michael A. Padlipsky, David W. Snow, and Paul A. Karger. Limitations of end-to-end encryption in secure computer networks. Technical Report ESD-TR-78-158, The MITRE Corporation: Bedford MA, HQ Electronic Systems Division, Hanscom AFB, MA, August 1978.

[39] The Tor Project. Tor: anonymity online. `http://www.torproject.org/`, 2009.

[40] T. Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, and Tadayoshi Kohno. Devices that tell on you: privacy trends in consumer ubiquitous computing. In *Proceedings of 16th USENIX Security Symposium*, pages 1–16, 2007.

[41] Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 61–74, New York, NY, USA, 2009. ACM.

[42] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the 10th conference on USENIX Security Symposium*, pages 25–25, 2001.

[43] François-Xavier Standaert, Tal G. Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *EUROCRYPT '09: Proceedings of the 28th Annual International Conference on Advances in Cryptology*, pages 443–461, Berlin, Heidelberg, 2009. Springer-Verlag.

[44] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, 2004.

[45] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 19, 2002.

[46] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[47] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 PLDI conference*, pages 32–41, New York, NY, USA, 2007. ACM.

[48] Mark Weiser. Program slicing. In *ICSE'81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[49] Charles V. Wright, Lucas Ballard, Scott E. Coull, Fabian Monrose, and Gerald M. Masson. Spot me if you can: Uncovering spoken phrases in encrypted voip conversations. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 35–49, 2008.

[50] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, Canada, 2009. USENIX Association.