Dynamic Taint Tracking in Managed Runtimes

Benjamin Livshits

Microsoft Research

Microsoft Research Technical Report

MSR-TR-2012-114

Abstract

This paper provides a taxonomy of runtime taint tracking approaches for managed code, such as code written in Java, C#, PHP, Perl, or Ruby. It covers main applications of data tainting such as preventing web application vulnerabilities including cross-site scripting and SQL injection attacks, along with disallowing privacy-sensitive data leaks. In addition to giving an overview of related literature from the last decade, this paper provides guidance and describes the trade-offs of different instrumentation approaches. Lastly, we provide a list of open problems whose solutions would aid practical adaption of runtime tainting on a wider scale.

***Abstract*—This paper provides a taxonomy of runtime taint tracking approaches for managed code, such as code written in Java, C#, PHP, Perl, or Ruby. It covers main applications of data tainting such as preventing web application vulnerabilities including cross-site scripting and SQL injection attacks, along with disallowing privacy-sensitive data leaks. In addition to giving an overview of related literature from the last decade, this paper provides guidance and describes the trade-offs of different instrumentation approaches. Lastly, we provide a list of open problems whose solutions would aid practical adaption of runtime tainting on a wider scale.**

I. Introduction

Data tainting has a long history, going back to Denning's seminal work [14]. Much of the recent programming language research has focused on language-based techniques for managing both implicit and explicit information flow, primarily using static, typing-based mechanisms [47]. Information flow properties are usually formulated in terms of *non-interference*, a property of two program executions that expresses independence of (private) inputs. The focus on this paper is different: here we primarily focus on tracking *explicit* forward flow of runtime data, in an effort to monitor an easier to check and understand property of the current program run. The term runtime tainting arguably has its origins in the Perl language's taint mode [61]. Since then, a wide range of research has been done, on both implementing tainting in various systems and also using taint propagation as a building block for achieving other goals, i.e., as part of a symbolic execution system [29, 49].

This paper represents an attempt to summarize the last decade of research in runtime data tainting in *managed*, or memory-safe runtimes, i.e. those associated with Java/JVM, C#/.NET, JavaScript, PHP, Ruby, etc. A previous survey by Schwartz *et al.* focuses on taint in the *native*, binary context [53]. Binary-level tainting has a number of significant differences with what is presented in this paper.

In particular, granularity at which data can be addressed is generally lower, leading to the ability to reason about data at the level of individual bits, but also creating an *impedance mismatch* for programs written in languages other than C and C++. Indeed, how does one express the fact that sanitization function `filter_var` has been called with parameter `FILTER_VALIDATE_INT`? What combinations of `EAX` and `EBX` registers does that correspond to? Tainting at the level of managed runtimes is generally closer to how developers think about their code, in terms of objects, methods, and variables, rather than in terms of low-level hardware details.

Of course, the very problems that have sparked interest in the area of runtime tainting have also been different. While native taint propagation has focused on detecting and preventing memory vulnerabilities such as buffer overruns [9, 24], much of the interest between 2005–2010 has come from the need to track *integrity* violations in web applications which lead to cross-site scripting attacks (XSS) [11, 46, 55] and SQL injection attacks [1, 17, 30], as well as several other forms of injection attacks. More recently, starting around 2010, confidentiality violations or *privacy leaks*, especially in the context of mobile devices, have generated renewed interest in data tainting [15].

While now firmly a part of the dynamic analysis "toolkit", unlike other runtime security technologies such as stack canaries [2] or ASLR and DEP/NX [62], runtime tainting has not seen wide deployment outside of academia. In fact, we only know of one recent case of commercial deployment, in the context of the Fortify runtime security product; their technique is similar to the library instrumentation technique of Chin *et al.* [12]. We conjecture that this lack of wide commercial deployment is in part because of various deployment challenges posed by runtime tainting, combined with runtime overhead issues.

A. Contributions

This paper pursues the following broad goals:

- First, we attempt to pull together and classify most of the research literature in this space, providing a summary of work done thus far and assessing the state of experimental practice.
- Second, this paper tries to provide prescriptive guidance for someone trying to build a runtime taint tracking system.
- Third, we formalize the essence of dynamic taint tracking using an operational semantic, an approach previously used for taint tracking in a native setting [53].
- Fourth, we point out that performing dynamic tainting efficiently and precisely is far from a solved problem. We outline some of the remaining challenges.

B. Paper Organization

Section II provides a basic overview of how runtime tainting systems are built and summarizes some of the main applications of these techniques. Section III provides a formal definition of what many runtime taint tracking systems try to accomplish in the form of an operational semantics. Section IV proposes a taxonomy for classifying runtime tainting approaches and proceeds to categorize 17 projects using this taxonomy. Despite the abundance of research, practical deployment of runtime tainting has been somewhat spotty. We describe some of the reasons for this in Section V. Performance is a common stumbling block in both building and deciding to deploy a runtime tainting system; Section VI gives a broad performance comparison of existing methods. Section VII talks about optimizations designed to improve the overhead of instrumentation. Section VIII lists some of the major open problems in this space. Finally, Section IX provides our conclusions.

```
NODE 1 WPOBJ, WP_LOC 1938219150 System.Device.Location.GeoPosition‘1 52,34,16 266076234
NODE 2 WPOBJ, WP_LOC 354792480 System.Device.Location.GeoCoordinate 52,34,16 266076234
OBSV 2 [Param_CalleeSide:Void set_CurrentLocation(System.Device.Location.GeoCoordinate) 53,52,34,16 266076234]
OBSV 2 [FieldStore:STATIC-350-63464 53,52,34,16 266076234]
OBSV 2 [FieldLoad:STATIC-350-63464 202,201 266076234]
OBSV 2 [Return_CalleeSide 202,201 266076234]
OBSV 2 [FieldLoad:STATIC-350-63464 214,206,201 266076234]
OBSV 2 [Return_CalleeSide 214,206,201 266076234]
OBSV 2 [Param_CalleeSide:System.Uri GetListingRequest(System.Device.Location.GeoCoordinate, ...
OBSV 2 [Param_CalleeSide:System.Uri GetListingRequest(System.Device.Location.GeoCoordinate, ...
NODE 11 WPOBJ, WP_LOC -1973456913 System.Double 222,220,206,201 266076234
LINK 2 11 System.Device.Location.GeoCoordinate.get_Latitude
NODE 12 WPOBJ, WP_LOC -1619338289 System.Double 222,220,206,201 266076234
LINK 2 12 System.Device.Location.GeoCoordinate.get_Longitude
NODE 13 WPOBJ, WP_LOC -1578306455 System.String 222,220,206,201 266076234
LINK 11 13 System.String.Format
...
LINK 15 20 System.Object.ToString
NODE 21 WPOBJ, WP_LOC 2115030139 System.UriBuilder 222,220,206,201 266076234
LINK 20 21 System.UriBuilder.set_Query
NODE 22 WPOBJ, WP_LOC 2115030139 System.Uri 222,220,206,201 266076234
LINK 21 22 System.UriBuilder.get_Uri
OBSV 21 [Return_CalleeSide 222,220,206,201 266076234]
OBSV 21 [Return_CalleeSide 220,206,201 266076234]
OBSV 21 [Param_CallerSide:System.Net.WebClient.DownloadStringAsync 206,201 266076234] INTERNET
```

Fig. 1. Sample output of runtime instrumentation. For brevity, 14 records in the middle are omitted.

II. Overview

A. Constructing a Runtime Tainting System

In its most basic form, constructing a runtime tainting system involves the following basic steps:

1) decide on the level of instrumentation such as source-level, bytecode-level, runtime-level, etc.
2) identify relevant instrumentation points including *sources*, *sinks*, *sanitizers*, and *propagators* within the application to be instrumented and relevant libraries;
3) instrument at those points to record relevant information such as the object IDs that pass through the instrumentation point, thread ID, source type, etc.

By way of example, Figure 1 shows some sample output from a runtime instrumentation system in the course of a short run of a smartphone mobile app written in .NET. There are three types of records being shown: `NODE`, `LINK`, and `OBSV` records. `NODE` indicate the creation of relevant tainted objects, which result from calls to methods such as `GeoCoordinate.get_Latitude` (this method acts as a source because it surfaces privacy-sensitive data about the user's location to the rest of the application). `LINK` nodes correspond to *propagation* of taint from one object to another. Extra record fields show that they are often a result of string concatenation or appending characters to a `StringBuilder` object. Finally, `OBSV` records are observed uses of objects in methods of interest, such as sink calls. Figure 2 shows a graph representation of taint propagation in a mobile apps, demonstrating how GPS coordinates flow from the return of standard library method `System.Device.Location.GeoCoordinate` to a network sink.

B. Applications of Dynamic Tainting

While in the 1990s the interest in dynamic tainting had been primarily fueled by the desire to prevent memory errors such as buffer-overruns that may be exploitable with a carefully crafted input buffer, with managed languages and runtimes, we have seen two primary applications for runtime tainting, focusing on integrity and confidentiality properties.

- **Preventing injection attacks (integrity)** involves preventing SQL injections and cross-site scripting attacks (XSS) in web application. At their core, these vulnerabilities and their many variations in web applications (such as command injection, path injection, XSRF, etc. [43]) involve the propagating untrusted (and unsanitized or not properly sanitized) user data from a *source* to a sensitive *sink* within the application or one of its libraries [20, 31, 37].
- **Privacy leak prevention (confidentiality)** is a problem that has become especially relevant with the recent popularity of smart phone apps. It involves tracking sensitive user data (emanating from a source) leaving the application through a network, file system, or another similar sink [15, 22].

These two properties are in many ways duals of each other and similar techniques are used to track both, although we are not aware of projects that attempt to achieve both of these goals at the same time, in part because these properties largely pertain to different domains: web applications and mobile apps.

It is important to note that frequently runtime data tainting is used as a building block in the context of another technique. A primary example of this is using tainting as part of *symbolic execution*, a runtime path

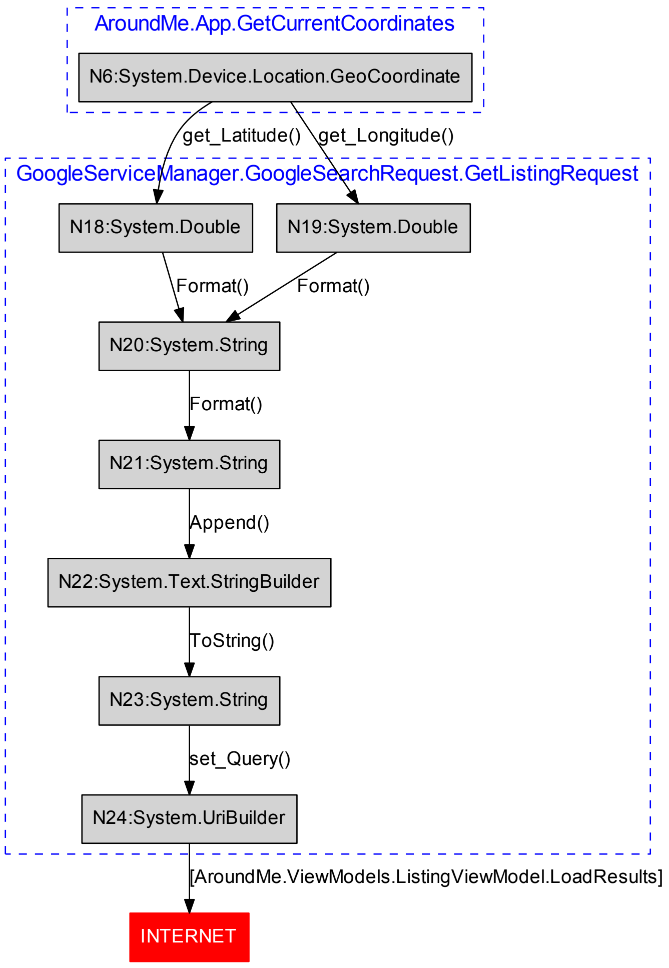
AroundMe.App.GetCurrentCoordinates
N6:System.Device.Location.GeoCoordinate
get_Latitude()
get_Longitude()
GoogleServiceManager.GoogleSearchRequest.GetListingRequest
N18:System.Double
N19:System.Double
Format()
Format()
N20:System.String
Format()
N21:System.String
Append()
N22:System.Text.StringBuilder
ToString()
N23:System.String
set_Query()
N24:System.UriBuilder
[AroundMe.ViewModels.ListingViewModel.LoadResults]
INTERNET

Fig. 2. Taint propagation in a mobile app showing flow from GPS location acquisition to the network.

exploration technique [26, 29, 49]. Runtime data tainting is used to determine which data is user-controlled and is therefore dangerous in the context of injection attack possibilities.

C. Motivating Examples

Several examples in this section provide intuition for much of the rest of the paper.

Example 1 Basic instrumentation. As illustrated in Figure 3, for a web application, typical sources include return results of methods like `HttpRequest.GetParameter` and `HttpRequest.GetHeader` in C# or `HttpServletRequest.getParameter` in Java. In PHP, returned `$_GET` array elements are treated as sources.

Two common application-level instrumentation approaches are common: source- and bytecode-level instrumentation. The simple C# code snippet in Figure 3 shows the difference between source-level and bytecode-level instrumentation.

The disassembled version in Figure 4 tells a much clearer story, effectively desugaring idioms like `Params["name"]`

```
void ProcessRequest(HttpRequest request,
                    HttpResponse response)
{
    string s1 = request.Params["name"];
    string s2 = request.Params["encoding"];

    response.Output.WriteLine("Parameter␣" + s1);
    response.Output.WriteLine("Header␣" + s2);
}
```

Fig. 3. Simple reflective cross-site vulnerability.

and concatenation using the `+` operator. Indeed, `Params` map references correspond to calls to `get_Item` on lines 11 and 16. The `+` operator is desugared into `String.Concat` calls on lines 22 and 29. At the bytecode level these calls are in fact easy to match. ■

Example 2 PHP Taint. A taint mode for the PHP interpreter proposed by Wietse Venema of IBM Reserch [60] supports the following taint "flavors": `TC_HTML`, `TC_SHELL`, `TC_MYSQL`, `TC_MYSQLI`, `TC_PCRE`, and `TC_SELF` to represent HTML output, shell command arguments, MYSQL query parameters, MYSQLI query parameters, regular expression patterns, and `eval` parameters, respectively. A careful read of the PHP taint proposal suggests the policy table shown in Figure 5. Unfortunately, for many systems the policy table is often implicit or is not fully specified. In this case, no clear guidance is given for data from the database being sent to `eval`. ■

Example 3 Encrypted cloud. While the discussion so far has focused on integrity properties, similar runtime instrumentation machinery can be employed for confidentiality. Indeed, consider a web application using a public cloud provider for storage. The web application may want to use the cloud for scalability and to reduce storage hardware costs, but does not fully trust the cloud to protect the confidentiality of its data. The application therefore will use encryption when serializing data to the database, and decryption when deserializing.

	output	**cloud**
input	$\perp$	encrypt
cloud	decrypt	$\perp$

In this scenario, the sources are of types input and cloud and sinks are of types outside/browser and cloud. The policy would encrypt data before it goes into the cloud and decrypt it on the way out of the cloud. The correct processing to apply (if any) depends on both the source and sink type, as captured by the table above. ■

III. Formalization

This section aims to formalize the notion of a common explicit taint propagation policy and then proceeds to give an operational semantics capturing an implementation of taint tracking.

```
.method private hidebysig instance void ProcessRequest(class System.Web.HttpRequest request,
                                          class System.Web.HttpResponse response) cil managed
{
    .maxstack 3
    .locals init (
        [0] string s1,
        [1] string s2)
    L_0000: nop
    L_0001: ldarg.1
    L_0002: callvirt instance class NameValueCollection System.Web.HttpRequest::get_Params()
    L_0007: ldstr "name"
    L_000c: callvirt instance string NameValueCollection::get_Item(string)
    L_0011: stloc.0
    L_0012: ldarg.1
    L_0013: callvirt instance class NameValueCollection System.Web.HttpRequest::get_Params()
    L_0018: ldstr "encoding"
    L_001d: callvirt instance string NameValueCollection::get_Item(string)
    L_0022: stloc.1
    L_0023: ldarg.2
    L_0024: callvirt instance class System.IO.TextWriter System.Web.HttpResponse::get_Output()
    L_0029: ldstr "Parameter␣"
    L_002e: ldloc.0
    L_002f: call string System.String::Concat(string, string)
    L_0034: callvirt instance void System.IO.TextWriter::WriteLine(string)
    L_0039: nop
    L_003a: ldarg.2
    L_003b: callvirt instance class System.IO.TextWriter System.Web.HttpResponse::get_Output()
    L_0040: ldstr "Header␣"
    L_0045: ldloc.1
    L_0046: call string System.String::Concat(string, string)
    L_004b: callvirt instance void System.IO.TextWriter::WriteLine(string)
    L_0050: nop
    L_0051: ret
}
```

Fig. 4. Disassembled C# code snippet explicitly showing relevant calls.

	HTML output	**shell command arguments**	**MYSQL query parameters**	**MYSQLI query parameters**	**regular expression patterns**	**eval parameters**
Web	`htmlspecialchars()`	`escapeshellcmd()`	`mysql_escape_string()`	`mysqli_escape_string()`	`preg_quote()`	`untaint($var, TC_SELF)`
Database	`htmlspecialchars()`	`escapeshellcmd()`	`mysql_escape_string()`	`mysqli_escape_string()`	`preg_quote()`	???

Fig. 5. PHP taint sanitization policy table.

$\langle program \rangle ::= \langle statement \rangle^*$

$\langle statement \rangle\ s ::=$
- $|\ \langle var \rangle$ '=' $\langle expr \rangle$
- $|\ \langle var \rangle$ '=' $\langle var \rangle.\langle field \rangle$
- $|\ \langle var \rangle$.field '=' $\langle var \rangle$
- $|\ \langle var \rangle$ '=' `alloc`
- $|\ \langle var \rangle$ '=' $\mathcal{I}_k$
- $|\ \langle var \rangle$ '=' $\mathcal{P}_k(\langle var \rangle)$
- $|\ \langle var \rangle$ '=' $\mathcal{S}_k(\langle var \rangle)$
- $|\ \mathcal{O}_k(\langle var \rangle)$
- $|$ '`if`' $\langle expr \rangle$ '`then`' $\langle statement \rangle$ '`else`' $\langle statement \rangle$
- $|$ '`while`' $\langle expr \rangle$ '`do`' $\langle statement \rangle$
- $|$ `skip`

$\langle expr \rangle\ e ::=$
- $|\ \langle expr \rangle \diamond \langle expr \rangle$
- $|\ \langle var \rangle$
- $|\ \langle const \rangle$ | `true`| `false`
- $|$ `null`

Fig. 6. BNF grammar for our language. $\diamond$ indicate typical binary operators. *const* represent numeric and string constants.

A. Taint Propagation

Although this is not often made explicit, traditionally, taint propagation policies have followed a certain shape, as described in the examples above. While we shall talk about sanitizers, it should be understood that for privacy-focused applications, declassifiers may be used in their stead.

Definition 1: An explicit taint propagation problem Π consists of the following five-tuple $\Pi = \langle \mathcal{I}, \mathcal{O}, \mathcal{P}, \mathcal{S}, \mathcal{T} \rangle$:

- Sources $\mathcal{I} = \{\langle m, t \rangle, \ldots\}$
- Sinks $\mathcal{O} = \{\langle m, t, h \rangle, \ldots\}$
- Propagators $\mathcal{P} = \{\langle m, t_1, t_2 \rangle, \ldots\}$
- Sanitizers $\mathcal{S} = \{\langle m, t_1, t_2 \rangle, \ldots\}$
- Policy table $\mathcal{T} = \mathcal{I} \times \mathcal{O} \mapsto \langle s_1, s_2, \ldots \rangle$

where m is a method (or function) and taint labels $t_i \in \mathcal{T}$, which is a semi-lattice with element $\top$ representing (fully) untainted, are the taint *labels* used to distinguish between different kinds of taint sources and sinks. We also allow sink *handlers* that specify what to do in the case of a violation, i.e. insufficiently sanitized flow from a source to a sink; typically, these handlers will just print an error message and terminate the program.

We assume that aside from the specified side-effect on taint labels, functions m in sources, sanitizers, propagators, and sink do not have the ability to change taint labels.

$$\frac{\Sigma \vdash e \Downarrow \langle v, t\rangle \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \qquad \Sigma.\tau'_\Delta = \Sigma.\tau_\Delta[x \leftarrow t]}{(\Sigma, x = e) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Assign}$$

$$\frac{\Sigma \vdash \chi[\langle y, f\rangle] \Downarrow \langle v, t\rangle \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow v] \qquad \Sigma.\tau'_\Delta = \Sigma.\tau_\Delta[x \leftarrow t]}{(\Sigma, x = y.f) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Load}$$

$$\frac{\chi' = \chi[\langle x, f\rangle \leftarrow \Sigma.\Delta[y]] \qquad \tau'_\chi = \tau_\chi[\langle x, f\rangle \leftarrow \Sigma.\tau_\Delta[y]]}{(\Sigma, x.f = y) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Store} \qquad \frac{\Sigma.\Delta' = \Sigma.\Delta[x \leftarrow \texttt{null}] \qquad \Sigma.\tau'_\Delta = \Sigma.\tau_\Delta[x \leftarrow \top]}{(\Sigma, x = \texttt{alloc}) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Alloc}$$

$$\frac{\mathcal{I}_k = \langle m, l\rangle \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow \Sigma.\Delta[m()]] \qquad \Sigma.\tau'_\Delta = \Sigma.\tau_\Delta[x \leftarrow l]}{(\Sigma, x = \mathcal{I}_k()) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Source}$$

$$\frac{\mathcal{O}_k = \langle m, l, h\rangle \qquad \Sigma.\tau_\Delta[x] = \top}{(\Sigma, \mathcal{O}_k(x)) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Sink} \qquad \frac{\mathcal{O}_k = \langle m, l, h\rangle \qquad \Sigma.\tau_\Delta[x] \neq \top}{(\Sigma, \mathcal{O}_k(x)) \rightsquigarrow (\Sigma', h)}\ \textsc{Sink-Fail}$$

$$\frac{\mathcal{S}_k = \langle m, t_1, t_2\rangle \qquad t_1 = \Sigma.\tau_\Delta[y] \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow m(\Sigma.\Delta[y])] \qquad \Sigma.\tau'_\Delta = \Sigma.\tau_\Delta[x \leftarrow t_2]}{(\Sigma, x = \mathcal{S}_k(y)) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Sanitizer}$$

$$\frac{\mathcal{P}_k = \langle m, t_1, t_2\rangle \qquad t_1 = \Sigma.\tau_\Delta[y] \qquad \Sigma.\Delta' = \Sigma.\Delta[x \leftarrow m(\Sigma.\Delta[y])] \qquad \Sigma.\tau'_\Delta = \Sigma.\tau_\Delta[x \leftarrow t_2]}{(\Sigma, x = \mathcal{P}_k(y)) \rightsquigarrow (\Sigma', \texttt{skip})}\ \textsc{Propagator}$$

$$\frac{}{(\Sigma, \texttt{skip};\, S) \rightsquigarrow (\Sigma, S)}\ \textsc{Skip} \qquad \frac{(\Sigma, S_1) \rightsquigarrow (\Sigma', S'_1)}{(\Sigma, S_1;\, S_2) \rightsquigarrow (\Sigma', S'_1;\, S_2)}\ \textsc{Sequence}$$

$$\frac{\Sigma \vdash e \Downarrow \langle \texttt{true}, t\rangle}{(\Sigma, \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \rightsquigarrow (\Sigma, S_1)}\ \textsc{If-T} \qquad \frac{\Sigma \vdash e \Downarrow \langle \texttt{false}, t\rangle}{(\Sigma, \texttt{if}\ e\ \texttt{then}\ S_1\ \texttt{else}\ S_2) \rightsquigarrow (\Sigma, S_2)}\ \textsc{If-F}$$

$$\frac{\Sigma \vdash e \Downarrow \langle \texttt{true}, t\rangle}{(\Sigma, \texttt{while}\ e\ \texttt{do}\ S) \rightsquigarrow (\Sigma, \texttt{while}\ e\ \texttt{do}\ S)}\ \textsc{While-T} \qquad \frac{\Sigma \vdash e \Downarrow \langle \texttt{false}, t\rangle}{(\Sigma, \texttt{while}\ e\ \texttt{do}\ S) \rightsquigarrow (\Sigma, \texttt{skip})}\ \textsc{While-F}$$

$$\frac{\Sigma \vdash e_1 \Downarrow \langle v_1, t_1\rangle \qquad \Sigma \vdash e_2 \Downarrow \langle v_2, t_2\rangle \qquad e = v_1 \diamond v_2 \qquad t = t_1 \sqcap t_2}{\Sigma \vdash e_1 \diamond e_2 \Downarrow \langle v, t\rangle}\ \textsc{BinOp} \qquad \frac{}{\Sigma \Downarrow \langle \Sigma.\Delta[\mathit{var}], \Sigma.\tau_\Delta[\mathit{var}]\rangle}\ \textsc{Var}$$

$$\frac{}{\Sigma \vdash v \Downarrow \langle v, \top\rangle}\ \textsc{Const} \qquad \frac{}{\Sigma \vdash v \Downarrow \langle \texttt{null}, \top\rangle}\ \textsc{Null} \qquad \frac{}{\Sigma \vdash v \Downarrow \langle \texttt{true}, \top\rangle}\ \textsc{True} \qquad \frac{}{\Sigma \vdash v \Downarrow \langle \texttt{false}, \top\rangle}\ \textsc{False}$$

Fig. 7. Operational semantics.

We assume that handlers do not change taint labels either. However, as discussed in Section V-E, it is possible to do more aggressive kind of recovery. Finally, the policy table provides the appropriate list of sanitizers to apply for each source/sink pair, as shown the the example in Figure 8.

B. Operational Semantics for a Small Language

Figure 6 gives a BNF grammar for a small Java-like language that captures the essence of what is needed to explain the tracking of taint. Note that we have explicit statements for reading input from a source, and also invoking sanitizer, propagator, and sink methods with parameters. Note that we assume that temporaries have been inserted to use variables where expression would typically be used otherwise.

	$\mathcal{O}_1$	$\mathcal{O}_2$	$\mathcal{O}_3$
$\mathcal{I}_1$	$\mathcal{S}_1$	$\mathcal{S}_1$	$\mathcal{S}_4$
$\mathcal{I}_2$	$\mathcal{S}_1$	$\mathcal{S}_2$	$\mathcal{S}_2$
$\mathcal{I}_3$	$\mathcal{S}_2$	$\mathcal{S}_1$	$\mathcal{S}_3$

Fig. 8. Example policy. Sources shown vertically; sinks shown horizontally.

In a manner similar to that of Schwarts *et al.* [53], we proceed to define an operational semantics for dynamic explicit taint tracking. The execution context is described by the following parameters contained within Σ: Δ, which maps a variable name to its value; τ_Δ, which maps a variable name to its taint label; χ, which maps a heap object field to its value; τ_χ, which maps a heap object field to its taint label. We do not assume anything about how labels are represented or stored: they can be contained within object headers or be stored completely on the side. Moreover, the level of detail contained in labels can vary from being a one-bit tainted/untainted representation to a more elaborate lattice that distinguishes different kinds of taint sources.

Furthermore, we shall use x, y, z to denote variables $\langle \mathit{var}\rangle$, and f to denote fields $\langle \mathit{field}\rangle$. Here are a few examples: $\Sigma.\Delta[x]$ gives the current value of variable x; $\Sigma.\chi[\langle x, f\rangle]$ gives the current value of variable $x.f$. Map updates are denoted with $\leftarrow$: for instance, $\Sigma.\Delta[x \leftarrow \texttt{hello}]$ represents an updated $\Sigma.\Delta$ map with x set to string `"hello"`.

Figure 7 shows operational semantic inference rules for a typical implementation of dynamic taint tracking. Several rules in particular require a discussion.

- SANITIZER: This rule acts to transform taint labels from t_1 to t_2 as long as the input label matches the label expected by sanitizer $\mathcal{S}_k$.
- SINK: Expects the label of its argument to be $\top$.
- SINK-FAIL: Calls the sink handler h if its argument does not have label $\top$.
- CONST: We initialize the taint label of constants to $\top$ or "not tainted." Same is true of the results of `alloc` calls.
- BINOP: We apply the lattice meet operator $\sqcap$ to combine two labels from the left and right hand sides.

Property 1 (*sound taint tracking):* for every runtime value $v = \langle e, t\rangle$:

- p is fully untainted, i.e. $t = \top$; or
- p is returned from a call to m of source of the form $I_k = \langle m, t\rangle$; or
- the *last* sanitizer/propagator returning v applied was of the form $\langle m, t_0, t\rangle$.

Intuitively, this property captures correctness of taint tracking. This property holds since the only inference rules that change the taint labels on values are SANITIZER and PROPAGATOR rules and it is not possible to manufacture of forge labels in our semantics.

IV. TAXONOMY

This section proposes a taxonomy of choices for a runtime taint propagation system. Figure 11 classifies 17 projects according to this taxonomy.

A. Level of Instrumentation

Runtime taint tracking can be implemented at several levels, affecting the instrumentation precision, overhead, and level of implementation difficulty. At a high level, we distinguish between *application-level* and *system-level* instrumentation. Below we outline some of the trade-offs. Figure 9 provides more prescriptive guidance as to which method to choose.

Source-level instrumentation: is an attractive possibility because it requires relatively little infrastructure support other than a language parser. This option is frequently used for scripting languages such as JavaScript, where the source code is readily available [25, 27, 39]. Programs instrumented at the source level are easier to debug and understand. Some shortcomings involve the need to capture all syntactic constructs that correspond to a particular semantic operation. For instance, if we would like to instrument all variable assignments, we will need to consider simple syntactic forms such as $\texttt{x} = \texttt{y}$, but also interprocedural assignments of actual arguments to formal ones, as well as less obvious assignments resulting, for instance, from having to properly instrument loop initialization constructs of the form $\texttt{for(i = 0; i < 100; i++)}$.

Bytecode-level instrumentation: is similar to source-level instrumentation but often is more challenging in practice [15, 37]. Part of the problem is the need to produce instrumented code that is deemed to be valid, typically according to a bytecode verifier such as those found in Java and C#.

This might require worrying about balancing the stack, creating parameters of the right type, etc. Other challenges include instrumenting built-in core types such as `Object` in Java or `System.Type` in .NET. Just like with source-level rewriting, another challenge with this technique is the difficulty of instrumenting dynamically-loaded code. Some of the runtime instrumentation frameworks overcome this by allowing the user to register a callback invoked each time a library is loaded dynamically or a new piece of code is fetched in the source form (to be passed to `eval`, for instance).

Other difficulties not present with source-level instrumentation involve the need to potentially re-sign the bytecode and repackage it into a JAR file, a DLL, etc.

Of course, bytecode-level instrumentation is often the only option is there is not access to source code, which is frequently true for large projects that rely on libraries.

Library-level instrumentation: is a useful alternative to bytecode-level instrumentation, especially for JVM and .NET, assuming one can rewrite and re-deploy (standard) libraries. This is the the approach chosen in Chin [12] and the Fortify runtime analysis tool [16]. Indeed, in the extreme case, if *all* sources, sinks, sanitizers, and propagators are library methods, we can perform the majority of instrumentation within the library, without touching the application code at all.

The main advantage of this techniques is a frequently observed reduction in the runtime overhead. The disadvantage is that tweaking with standard libraries often reduces the stability of application execution, as it violates unstated invariants that the runtime expects. One has to be particularly careful when rewriting `Object` in Java or `System.Type` in .NET, but even slight modification to some methods of the `String` class can lead to surprising failures at runtime. There is generally no way to determine this ahead of time, other than testing for compatibility with the different runtimes and its versions, such as the many versions of JVM from Oracle, IBM, and other vendors, .NET runtimes on different platforms and operating systems, etc. The cost of better performance with this approach is its lesser compatibility.

Another issue is the inability to completely restrict instrumentation exclusively to library code only. For instance, Halfond *et al.* report the need to instrument constant string creation within the application [21], which obviously creates fewer instrumentation points than full

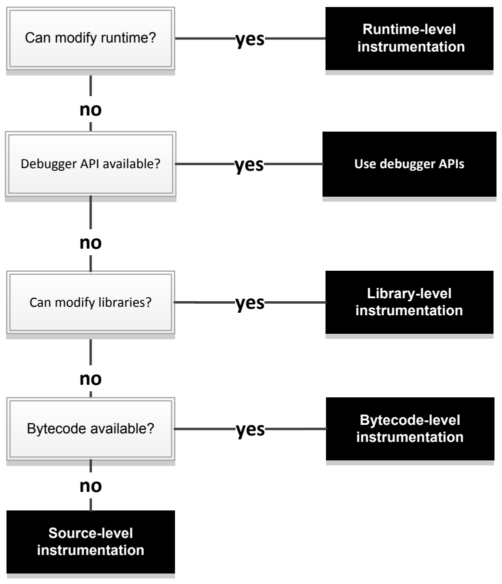

Fig. 9. Where to instrument?

bytecode-based instrumentation, but still perturbs application code, which might be undesirable for deployment because this increases code size, etc.

Debugging APIs: while this option is not always available, widely-used runtimes such as JVM (JavaTM Virtual Machine Tool Interface) and .NET (.NET Framework debugging and profiling APIs) provide useful facilities for monitoring and controlling program execution. The advantage of this approach is its ostensible simplicity: we get access to low-level runtime internals without having to rebuild the runtime or having access to its sources. The disadvantage is the fragility of this approach and the limitations of debugging APIs. Moreover, these APIs can vary based on the version of the runtime, its vendor, and the operating system.

Runtime-level instrumentation: is often desirable because it providers the most complete insight into application execution, assuming one has access to runtime internals. A significant advantage of this approach is the low memory overhead: extra taint bits can frequently be stored in the object header, eliminating the need for extra lookup taint maps. For example, consider the Rubinius runtime for Ruby. The first 32-bit integer at the start of the object header contains flags about the object. Along with `Pinned`, `Frozen`, and other flags maintained by the runtime and the garbage collector, `Tainted` is the tainted bit for the object.

Note that deploying instrumented runtimes is not always desirable: they might conflict with the "default" runtime installed on the machine, buggy instrumentation will compromise the ability to run other applications, etc. Several runtimes have adopted built-in taint tracking, including Perl [61], PHP [60], Ruby, and JavaScript. However, JavaScript quickly rolled back their taint support, which used to be part of JS1.1 in Netscape 3 in 1996.

Summary: Loosely, we can describe source- and bytecode-level instrumentation as application-level instrumentation and other approaches as system-level instrumentation. While it is often desirable, it is not always possible to restrict ourselves to system-level instrumentation only. Positive tainting mentioned above is one reason. Another is the fact that it is frequently desirable to track taint as it passes through local primitive values. For example, to track GPS location provenance in .NET, we need to instrument primitive numeric value manipulation at the level of a method. To summarize the trade-offs above, Figure 9 provides a decision diagram for choosing the level of instrumentation for a tool one intends to build.

B. Tracking Granularity

Tracking strings: Much attention in building runtime tainting systems has been given to tracking string data as it passes through the application. The level at which taint is propagated through the application varies depending on the approach. Note that unlike instrumentation in native applications, byte-level tainting is too low-level and is consequently uncommon. Several alternative approaches have been proposed.

- *character-level*: given that taint propagation involves following strings around much of the time, maintaining taint data at the level of individual characters has the advantage of enabling more precise analysis.
- *modeling strings*: it has been proposed that taint can be tracked through sanitizers and that strings can be modeled more accurately [19, 49, 50].
- *object-level*: a more coarse-grained approach is to taint individual (string) object.

Tracking (other) primitive types: While generally a non-issue for injection attacks such as XSS, tracking primitive types becomes important when dealing with confidential (numeric) data such as GPS location coordinates or the user's annual income. Because runtimes typically manipulate primitive data differently from strings or objects, tracking the propagation of this type of data is considerably more involved. For example, in .NET, C# statements `int x = 43; int y = x;` result in the following instructions, where the `.locals` block mentions two locals at offsets 0 and 1 corresponding to `x` and `y`. These locals can be manipulated using instructions such as `stloc` and `ldloc`.

```
.locals init ([0] int32 x, [1] int32 y)
    L_0000: nop
    L_0001: ldc.i4.s 0x2b
    L_0003: stloc.0
    L_0004: ldloc.0
    L_0005: stloc.1
```

Tracking collections: The granularity with which data placed in collections (which includes arrays and maps)

```
public class GeoLocation {
  double lattitude;
  double longitude;

  public GeoLocation(double lattitude, double longitude){
     this.lattitude = lattitude; this.longitude = longitude;
  }

  public void setLattitude(double lattitude) {
      this.lattitude = lattitude;
  }
  public void setLongitude(double longitude) {
      this.longitude = longitude;
  }
}
```

Fig. 10. `GeoLocation` example.

should be tracked requires careful consideration. The common approach of tainting the entire collection suffers from the loss of precision is introduces: anything subsequently retrieved from that collection will be marked as tainted. Luckily, *objects* placed in collections can be tracked individually without any loss of precision or need to taint the collection; the same is not true of, for example, an array of potentially tainted floating point values. A more precise approach involves keeping array indices of tainted array elements. In the case of languages interacting with the web document DOM such as JavaScript, tracking taint within DOM elements represents a particularly acute problem. A potential solution involves preventing tainted data from reaching the DOM, i.e. treating write access to the DOM as a sink.

Tracking (other) objects: Another challenge comes from having to track non-collection objects that may have tainted data as their fields. A common example is shown in Figure 10. Should we treat `GeoLocation` as tainted if either `double` parameter to the constructor is tainted? While this is probably the right thing to do, correcting propagating taint to `GeoLocation` objects requiring adding the constructor as both setter methods to the list of taint propagators. Having to deal with objects such as this that have multiple tainted fields adds further complications in terms of representation of taint labels.

C. Form of Tainting

The most common form of tainting is *negative* tainting, which involves tainting untrusted sources and propagating this data to sinks for integrity and sensitive data as sources and data release points for confidentiality. An alternative approach is *positive* tainting, which taints *trusted* data, typically (string) constants and other application-controlled data. An example of positive tainting use comes from a paper by Halfond *et al.*, which proposes a method for preventing SQL injection attacks [21].

The key difference is that positive tainting effectively *white-lists* sources of safe data instead of *black-listing* safe data, which Halfond *et al.* perceive as a plus because it favors false positive over false negatives. Oddly, despite this challenge, we see relatively little work on positive tainting systems. This highlights the difficulty of coming up with a comprehensive *policy* for dynamic taint tracking, which is further described in Section V-A.

Focus on negative tainting can also be explained by the fact that negative tainting is generally useful "out of the box" — while it might not find every violation, it will not overwhelm the user with false positives, either. Developers and security engineers can refine the policy over time, whereas having to wade through dozens or hundreds of false positive alarms is virtually guaranteed to limit adaption.

D. Explicit vs. Implicit Flow

A useful distinction is between explicit or *direct* information flow, which occurs through copying values through the program and implicit flow, which commonly happens when the outcome of conditionals that depend on tainted data in the program can be discerned through the state of other values, as shown in the commonly used code snippet below.

```
if (confidential == 1) {
    public = 42
} else {
    public = 17;
    public = 0;
}
```

A great deal of work has been done on information flow tracking [47], especially in the static context, with some notable exceptions in the dynamic space [3, 4, 13]. Unfortunately, much work in the static space requires specialized static type systems and does not usually directly translate into runtime implementations that can apply to large legacy systems.

When it comes to practical runtime analysis of large, complex *benign* programs, explicit taint tracking is employed. Note that implicit flow is considerably more important for analyzing malware or untrusted third-party code, which can be easily rewritten by the attacker to conceal malicious taint transfer. We are only aware of one project that uses runtime analysis for tracking implicit flow [3]. Because of this, much of the rest of this paper focuses on tracking explicit flow.

V. Deployment Challenges

While the basic principles of taint propagation instrumentation and runtime tracking are easy to understand, there is a wide range of significant deployment challenges, some of which explain the dearth of commercially available systems in this space.

A. Policy Specification Difficulties

At the core of a dynamic taint propagation system is the task of selecting an appropriate policy. In fact, selecting the right policy is often as difficult if not more so than designing the instrumentation itself. The Merlin project provides statistics for a tool whose policy contains 27

Project	Year	Runtime	Runtime mechanism	Granularity	Positive vs. negative	Implicit vs. explicit
[42]	2005	PHP	runtime	character	negative	explicit
[18]	2006	Java	library	object	negative	explicit
[37]	2006	Java	bytecode	object	negative	explicit
[20]	2006	Java	library	character	positive	explicit
[21]	2006	Java	library	character	positive	explicit
[40]	2007	JavaScript	runtime	object	negative	explicit
[6]	2008	PHP	runtime	character	negative	explicit
[35]	2008	Java	bytecode	object	negative	explicit
[54]	2009	PHP	library	character	negative	explicit
[26]	2009	PHP	runtime	object	negative	explicit
[12]	2009	Java	library	character	negative	explicit
[3]	2009	λ_{info}	runtime	object	negative	implicit
[15]	2010	Java	runtime	object	negative	explicit
[49]	2010	JavaScript	runtime	character	negative	explicit
[50]	2010	JavaScript	runtime	character	negative	explicit
[8]	2011	PHP	runtime	character	negative	explicit
[7]	2012	Python	library	object	negative	explicit

Fig. 11. Projects employing dynamic tainting in managed languages and runtimes: a summary.

sources, 77 sinks, and 7 sanitizers [33]. "Tuning" the policy is a challenging task: for negative tainting, failing to include relevant sources will lead to missed flows.

Omitting sanitizers will lead to incorrectly unterminated flows. Missing sinks will lead to failing to flag erroneous flows. Of course, having too many sources, for example, is also a problem, as it will lead to the problem of taint spread, where too many objects are deemed tainted, typically leading to too many warnings, rendering resulting tools useless in practice. Policy selection can dramatically affect the performance overhead measurements as well: if the policy is too "sparse", the overheads will be predictably, but deceptively low.

Alas, it very difficult to formulate the "correct" policy in general, even with having in-depth knowledge of the underlying application. Anecdotally, commercial static analysis tools such as Fortify and Coverity take a considerable amount of effort to deploy of large legacy code bases; much of this effort goes into refining the policy. This fundamental challenge we think in part explains the lack of adoption: developing a useful policy can be both time-consuming but necessary before taint propagation yields any useful results. In practice, we often see developers or security engineers editing source and sink specification files. However, if it takes a great deal of customer efforts to deploy such a system, who can really afford to do so?

Relying on frameworks: One saving grace is that the policy is generally not entirely application-specific: there usually is a sensible general policy that is fitting for the underlying application framework. For instance, policies have been developed for J2EE servlets as well as more declarative web frameworks such as JSP, PHP, and ASP.NET. Methods in these frameworks can be used as sources and sinks. Built-in system methods can be used as propagators. Finally, some of these platforms provide built-in sanitizer libraries.

Recently, the idea of automatic sanitization has been advocated and in fact introduced in a range of widely-used frameworks. For instance, ASP.NET supports the concept of input *validators*, which can be attached to web page controls to limit possible inputs and report an error otherwise. The out-of-the-box validators are shown in Figure 12. Custom validators may be introduced through a `<asp:CustomValidator>` tag supported via developer-provided logic. While generally a boon for the developer, these validators, because they generally set a boolean `IsValid` flag, require additional challenges when tracking taint, as the control flow conditions of the program are now of crucial importance.

Interestingly, while validators are used for preventing integrity vulnerabilities, we are not aware of such attempts to introduce built-in easily supported declassifiers. Even simple declassification tasks such as hashing-out the digits of a social security number other than the last four have to be done by the developer.

Application-specific analysis parametrization: While relying on framework specifications is a helpful approach which provides an easy way to get started, in practice, one eventually discovers that a more detailed and accurate application-specific specification is often needed for the best results, both in terms of finding errors and avoiding false positives.

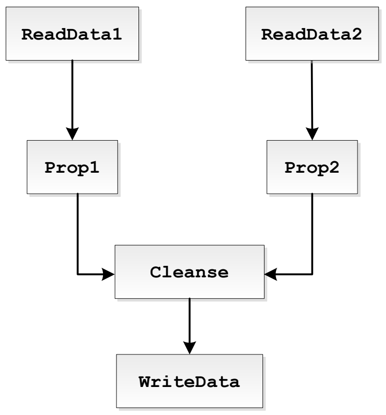

Fig. 13. Inferring sources, sinks, propagators and sanitizers with belief inference.

At the core of the approach advocated in the Merlin project [33] is the idea of belief inference: relying on the developer behaving correctly most of the time. Consider the example in Figure 13, showing interpredural data flow between different methods in the program.

Suppose we know for a fact that `ReadData1` is a source, `WriteData` is a sanitizer, `Prop1` and `Prop2` are propagators, and `Cleanse` is a sanitizer. We can then conclude with a high probability that `ReadData2` is another source. Indeed, why else would the developer sanitize the flow from `ReadData2` to `WriteData`? Suppose now that `ReadData1`

Rule to enforce	Validator	Description
Required entry	`RequiredFieldValidator`	Ensures that the user does not skip an entry.
Comparison to a value	`CompareValidator`	Compares a user's entry against a constant value, against the value of another control (using a comparison operator such as less than, equal, or greater than), or for a specific data type.
Range checking	`RangeValidator`	Checks that a user's entry is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates.
Pattern matching	`RegularExpressionValidator`	Checks that the entry matches a pattern defined by a regular expression. This type of validation enables you to check for predictable sequences of characters, such as those in e-mail addresses, telephone numbers, postal codes, and so on.
User-defined	`CustomValidator`	Checks the user's entry using validation logic that you write yourself. This type of validation enables you to check for values derived at run time.

Fig. 12. ASP.NET validators.

	HE1	HE2	HE3	HE4	OS1	OS2	OS3
HTMLEncode1	✓	✓	✓	0	–	✓	0
HTMLEncode2	✓	✓	✓	0	–	✓	0
HTMLEncode3	✓	✓	✓	0	–	✓	′
HTMLEncode4	0	0	0	✓	0	0	0
Outsourced1	–	–	–	0	✓	–	0
Outsourced2	✓	✓	✓	0	–	✓	0
Outsourced3	0	0	′	0	0	0	✓

Fig. 14. Equivalence matrix for seven implementations of HTML encoding. A ✓ indicates the implementations are equivalent. For implementations that are not equivalent, we show an example character that exhibits different behavior in the two implementations. The symbol `0` refers to the null character.

```
private static string EncodeHtml(string t)
{
    if (t == null) { return null; }
    if (t.Length == 0) { return string.Empty; }
    StringBuilder builder =
         new StringBuilder("", t.Length * 2);
    foreach (char c in t)
    {
      if ((((c > '`') && (c < '{')) ||
      ((c > '@') && (c < '['))) || (((c == ' ') ||
      ((c > '/') && (c < ':'))) || (((c == '.') ||
      (c == ',')) || ((c == '-') || (c == '_'))))){
        builder.Append(c);
      } else {
        builder.Append("&#" +
          ((int) c).ToString() + ";");
      }
    }
    return builder.ToString();
}
```

Fig. 15. Code for `AntiXSS.EncodeHtml` version 2.0.

and `ReadData2` are sources, `WriteData` is a sanitizer, and `Prop1` and `Prop2` are propagators. We can then conclude with a high probability that `Cleanse` is a sanitizer. The chances of this are enhanced by the fact that there are not one, but two source-to-sink paths that pass through method `Cleanse`.

Applying Merlin to 10 large business-critical Web applications that have been analyzed with CAT.NET, a state-of-the-art static analysis tool for .NET results in expanding the existing specification. A total of 167 new confirmed specifications were found, which result in a total of 322 additional detected vulnerabilities across the 10 benchmarks. More accurate specifications also reduce the false positive rate: Merlin-inferred specifications result in 13 false positives being removed, which constitutes a 15% reduction in the CAT.NET false positive rate.

B. Sanitizer Correctness

Much of the time, dynamic taint tracking assumes correct implementation of sanitizers (or declassifiers), effectively treating them like black boxes. Sanitizers are generally very difficult to implement correctly. Just like when it comes to home-crafted encryption routines, it is not recommended that application programmers implement sanitizers, because the chances of getting them wrong are exceedingly high. Much of the time, large development teams resort to using off-the-shelf sanitization libraries, either built into the underlying runtime, as is the case for basic sanitizers in JVM and .NET, or provided through a separate library such as AntiXSS now known as Microsoft Web Protection Library [38] or OWASP's ESAPI [44].

The problem of sanitizer correctness has been explored in the Bek project [57, 58]. Veanes *et al.* conduct an experiment asking developers to implement an HTML encoder based on an English description [58]. They then proceeded to compare these outsourced implementations `Outsourced1`–`Outsourced3` to off-the-shelf HTML encoders, summarizing the results in Figure 14. We discovered that `Outsourced1` escapes the `-` character, while `Outsourced2` does not. We also found that one of the `HTMLEncode` implementations does not encode the single quote character.

Because the single quote character can close HTML contexts, failure to encode it could cause unexpected behavior for a web developer who uses this implementation. For example, a recent attack on the Google Analytics dashboard was enabled by failure to sanitize a single quote [52].

Bek has been used to produce robust sanitizers with equivalent implementations in C, C#, and JavaScript, for both server- and client-side programming [59]. The current industrial practice involves a collection of libraries in different languages, which are at best loosely connected and encouraging developers to use these libraries to the best of their ability. There are, however, significant advantages to achieving parity in terms of sanitizers of different runtimes such as Java vs. JavaScript: code can be ported or migrated at runtime more easily, developers have clear guidance when they move from one platform to another.

C. Achieving Complete Mediation

One of the fundamental challenges to building a runtime tainting system is its *soundness*: how can we ensure that everything that requires instrumentation is indeed properly instrumented? Even if we believe the specification to be "complete", how to we ensure that we are not missing some relevant instrumentation points?

Fundamentally, instrumenting to achieve complete mediation is a more difficult goal that one might imagine because it requires statically matching runtime conditions. To see the difficulty, consider a call to `obj.toString()`. Must we instrument this call site? If this call may resolve to `String.toString()` or `StringBuilder.toString()`, then the answer is yes, as these are well-known propagator methods relevant for instrumenting Java programs. But how do we know what `obj.toString` may refer to? To answer this question, we would need to statically constrain the type of `obj`. But what if `obj` has been obtained from a collection, as illustrated by the Java code below.

```
StringBuffer buf = new StringBuffer();
for (Iterator iter = objects.iterator(); iter.hasNext();) {
   Object administrator = iter.next();
   buf.Append(obj.toString());
   buf.Append(' ');
}
```

Tracking the type and provenance of `obj` requires understanding where collection `objects` comes from and what objects might be put into it. This is general requires a whole-program pointer analysis or some other concrete type inference technique. This, however, is rarely done in practice, yielding an instrumentation approach that is either unsound, i.e., missing some relevant instrumentation points, or conservative. However, instrumenting every `Object.toString` call is likely to yield a very high number of instrumentation points and creates runtime overhead that is likely to be unacceptably high.

To evaluate the frequency of `Object.toString` calls, we took a 900 KB .NET DLL. We found 1 reference to `Object.toString` and 1 references to `String.toString` and 31 references to `StringBuilder.toString`. While not a common occurrence, this still represents about 3% of instrumentation points. The take-away here is similar to that in the case of static optimizations: completely sound instrumentation is very difficult to build.

D. Imprecision and Label Creep

Even runtime analysis, while generally considerably more precise than static analysis, is subject to precision challenges. Somewhat infamously, excessive taint label creep was the reason for the Netscape browser removing its support for taint within the JavaScript 1.1 runtime in 1996 [56], Chapter 34. Admittedly, Netscape attempted to support *implicit* taint by marking the PC as tainted, in an effort to enable cross-origin access to data with confidentiality support.

However, even explicit taint can suffer from a number of precision challenges. For object-oriented runtimes, it is possible to associate taint with a runtime object with a particular unique runtime identity. These unique identifiers might not be easy to establish; for example, object hash codes are not guaranteed to be unique. JVM, for example, offers the method `Object.hashCode()`. Typical implementations of `Object.hashCode()` involve computing a function of the allocated address of the object in memory, though this is not mandated by the standard. While the garbage collector can relocate the object, the value remains the same. While no uniqueness properties are provided, this method may be a good practical way to compute object identity, with hash collision for negative tainting leading to imprecision. For positive tainting, we can optimistically mark objects as safe, however, losing soundness.

Another common source of precision loss is interactions with external systems outside of the main runtime. For instance, the following code sets an attribute of a `DIV` DOM element to a tainted string `tainted`. Subsequently, a different portion of the code retrieves this attribute. Note

```
var tainted = ...;  // tainted string
var div = document.getElementById('id');
div.setAttribute('attr', tainted);
...
var elt = document.getElementByClassName('div');
var attr = div.getAttribute('attr');
```

that while `getElementById` is used for attribute setting, a different DOM API, `getElementByClassName` is used for retrieving the attribute. This example represents the common tension between loss of precision (mark every object coming out of the DOM as potentially tainted) and loss of soundness (mark every object coming out of the DOM as safe). How does the analysis resolve this tension?

Two common solutions include (1) treating "escaping" into the DOM as a sink, i.e. report runtime attempts to store tainted data into the DOM as warnings or errors; (2) creating and maintaining a finer-grained mapping at the DOM boundary, i.e. record that there's tainted data stored in attribute `attr` of element with ID `id`. Whenever a new attribute value is obtained from the DOM for attribute `attr`, check the ID of the underlying DOM element to see if there is a match, and proceed to mark it as tainted.

E. Automatic Sanitization

A key observation is that in large, complex code bases with non-trivial interprocedural flows, understanding whether a particular variable is tainted is a really complex task for the developer, even if they are familiar

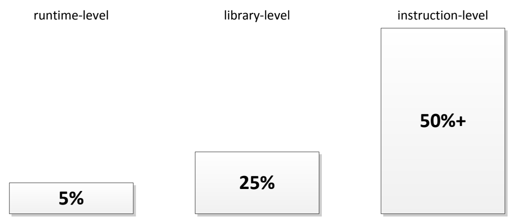

Fig. 16. Performance comparison.

	Android	TaintDroid	Overhead
App Load Time	63	65	3%
Address Book (create)	348	367	5%
Address Book (read)	101	119	18%
Phone Call	96	106	10%
Take Picture	1,718	2,216	29%

Fig. 17. TaintDroid overhead from [15].

with the code base. A series of recent efforts have focused on getting the developer out of the loop and automating the problem of sanitizer (or declassifier) placement.

SecuriFly project has proposed the idea of runtime compensation for injection flaws [37]. Instead of letting tainted data flow to a sink, it can be simply sanitized just before that happens. Of course, to do so, one needs to know what kind of sanitization to perform, which can be decided on the basis of the policy specification table $\mathcal{P}$ in Section III.

The challenges of manually performing proper sanitization are even more severe in large complex web applications that produce highly structured HTML output, such as, say, webmail systems. Saxena *et al.* explore the idea of runtime monitoring for avoiding injection-style security vulnerabilities in large and complex server-side applications [48, 51]. Key challenges emanate from the notion of *sanitization context.*

VI. Performance

Poor performance is the Achilles heel of runtime taint propagation. By way of providing general guidance, Figure 16 gives a rough estimate of performance overhead achieved with different approaches reported in the literature. Below we give several examples of reported overhead numbers.

Bytecode-level instrumentation: Martin *et al.* present the result of instrumenting sizable server-side Java applications to prevent SQL injection attacks. Their overhead numbers vary considerably, from 9–125% for unoptimized overhead and from less than 1% to 37% for when static analysis is used to eliminate unnecessary instrumentation points. However, the time to run static analysis is also quite considerable, despite the fact that it can be done offline and once, ranging from a little over a minute to about half an hour for the biggest application consisting of about 50,000 lines of Java code.

Library-level instrumentation: Chin *et al.* [12] provide an evaluation of this approach. Their overhead numbers range between 2 and 14%, however, these overheads are for small and potentially unrepresentative tasks. Their paper also reports throughput numbers, both for the original application and one instrumented with various levels of taint tracking.

Runtime-level instrumentation: Nguyen-Tuong *et al.* [42] describe a modified PHP interpreter designed to prevent injection attacks, reporting an overhead of less than 10%. In a related technical report [41] they provide micro-benchmark measurements by running individual PHP functions in a loop for 10,000 iterations. The highest measured overhead is 77% for the `sql.php` micro-benchmark which isolates the SQL injection checking. It creates a partially tainted string and passes it to the function that checks SQL commands. We believe that this is not representative of real performance and, indeed, overall performance numbers for three tasks (processing a login, entering a message and generating an output page from the contents of a database table) result in overheads of less that 5%.

TaintDroid reports a combination of overhead, both memory and time, for both micro-benchmark and real workloads. As mentioned before, the overhead ranges quite significantly depending on the task, to 29% on the high end. The absolute overhead for picture taking is about 0.5 seconds, which is certainly noticable by the end-user. The memory overhead tends to be considerably less pronounced.

Can we do better: While the papers mentioned above provide useful guidance, somewhat perplexingly, consistent and comprehensive reports describing performance of runtime tainting are hard to come by. We see several reasons for this.

- Part of the problem is the fact that especially for application-level instrumentation, overheads are very workload-specific. Indeed, the same web application may have a path that involves tainted data from a source quickly "dying off" resulting with very little overhead, and another path resulting in tainted data from a source traveling through the application and resulting in long and complex propagation chains. It is infrequent for runtime coverage numbers to be reported, so it is difficult to say whether reported overheads fall into the first or second category.
- In many cases, applications that benefit from runtime taint tracking are interactive ones, such as web-based applications and mobile phone apps. As such, traditional metrics of latency for a particular task are not very representative, indeed, the user is unlikely to notice a 10 ms slowdown.

Measuring performance differently: In this paper, we advocate a different way of both thinking about and reporting runtime overheads. Ultimately, we as a community would like to get to a point where runtime taint tracking can be always on. Four metrics that really matter:

1) reduction of throughput of large applications deployed in the cloud when the taint mode is on;
2) increase in the memory footprint as a result of extra "book-keeping" required for taint tracking;
3) increase in power consumption both on back-end servers and mobile devices;
4) increase in the code size, as more code means longer times to ship code (updates) to clients and worse code cache performance.

Currently, we are not aware of current large-scale deployed systems that provide this kind of evaluation.

VII. Optimizations

Using static analysis to reduce the amount of runtime instrumentation is a fairly tradition pairing of these analysis techniques. Of course, this approach suffers from soundness challenges: we want the underlying static technique to be sound to avoid removing relevant instrumentation point. However, designing a *fully* sound technique is tricky for most real languages. Indeed, it is almost impossible for a language as dynamic as JavaScript and is surprisingly tough even for a "well-behaved" language like Java, given reflection and dynamic code loading [10].

The problem of instrumentation overhead is hardly new when it comes to runtime taint tracking. A popular approach involves applying static analysis to minimize the amount of tracking that needs to occur at runtime. If we wish to maintain soundness of runtime tracking — a formidable challenge in the first place for a number of reasons described in Section V, we need to utilize a sound static analysis to perform this filtering. Below we outline several filtering approaches.

Type-based filtering: The simplest form of filtering involves using statically available or computed type information to minimize the number of instrumentation points. An example of this is the `+` binary operator used in many languages to indicate either numeric addition or string concatenation, depending on the type of the (first) argument. In a strongly-typed environment such as those provided by Java or .NET, there is no ambiguity between the numeric and string versions of the `+` operator; however, in dynamic languages such as JavaScript, further analysis of parameters is needed. RATA [34] provides an example of such an analysis, although the main focus is on the different numeric types and not as much on objects.

Forward and backward data slicing: A common approach to reducing the number of instrumentation points with application-based instrumentation involves only considering points that may both be reached from a source and may reach a sink [5, 23, 36]. Finding such points involves static interprocedural dataflow analysis, which for a language with pointers or references, arrays, and other data structures is both difficult to perform precisely

```
if(isSafe(input)){              string input2 = encode(input);
  stream.write(input);          stream.write(input2);
}
```

Fig. 18. Two ways to process untrusted inputs: checking (left) and rewriting (right).

and is also time-consuming. The PQL project applied this approach to reducing the taint tracking overhead for Java web applications [36]. The simple observation is that, given a policy specification, only certain parts of the program are relevant for taint propagation. This is similar to performing data slicing to minimize the relevant portion of the code.

Other techniques: A project by Livshits *et al.* [32] proposes a fully automatic sanitizer placement, which is done statically, whenever this is possible, "spilling" into runtime as infrequently as required. Of course, for complex real-life dataflow graphs, fully static sanitization is not always possible. The approach in that case is to automatically insert points where data is *tagged* and *untagged.* As an optimization goal, the duration of data tagging is minimized by starting to tag as late as possible and untagging the data as early as possible.

King *et al.* tackle a similar problem with a different approach [28]. They construct a graph representation of information flow in a program, such that source nodes are high-security inputs, and sink nodes are low-security outputs. A min-cut in this graph corresponds to a minimal set of program points that would allow the program to type-check.

VIII. Open Problems

The goal of this section is to highlight some of the open problems we see, both in terms of theoretical understanding of the problem and practical implementation issues.

Predictable performance: Historically, runtime mechanisms for security have received a lot of attention, but relatively few have been actually adopted in practice. "Lucky" cases include stack canaries [2] and the ASLR, DEP/NX suite of memory randomization and protection techniques [62]. The feature that distinguishes these technologies is their low and *predictable* performance overhead. An upper bound is perhaps even more desirable; in other words, a mean runtime overhead of 2% is often not good enough: a *maximum* of 5% is in fact considerably better.

Control flow tracking: The way we have been describing runtime taint tracking amounts to creating a dataflow propagation chain within a particular run. A propagation chain, however, is frequently not sufficient to establish a violation. The issue of *control dependence* comes up in the context of sanitization as well. Figure 18 shows two ways in which untrusted inputs can be manipulated. Our formulation in Section III supports the second approach of rewriting the tainted input. To support the first approach of input checking, we would need to be able to least partially track control dependencies. It is not entirely clear how to do so efficiently, as naïvely carrying the entire interprocedural control dependence around at runtime is bound to be costly.

Value tracking: What if we know that the particular data that is *currently* passed to a sink is safe, even if it deemed as potentially tainted by the runtime? For instance, it is possible to have an empty string that is tainted under our formulation. Should this be reported as a violation? How do we avoid doing so?

Declassifiers: While several widely-used encoder and sanitizer libraries exist, there is a lack of similar understanding of how declassifiers should be written. Is it not entirely clear whether there is room for standardized declassifier libraries or if the very task of declassification is fundamentally application-specific. If there is indeed room for such libraries, what functionality should be supported? Is writing (correct) declassifiers as difficult and error-prone as writing sanitizers? What are some of the correctness properties that are desirable? How can they be formulated?

Specification inference at runtime: While specification inference has been attempted in the static context [33], a system that continuously infers and enriches the specification as the application is executing would be very valuable. The advantage is the precision of runtime monitoring, combined with the lack of need to "train" the system beforehand.

Of course, the results of specification inference may be used in both a static and dynamic analysis context. Moreover, specifications inferred in the static context can be applied in a runtime taint propagation tool. We are not aware of attempts to perform specification inference based on runtime monitoring. Doing so would remove the imprecision inherent in a statically computed representation. To adopt the intuition from Merlin explained above, a natural approach would consider taint-carrying dataflow paths that have been sanitized that do not lead to a sink to discover candidate sinks or propagators.

Configurable runtime support for taint: While Perl and other scripting languages in many ways were ahead of the wave in natively supporting taint propagation systems, there is a lack of support for even basic forms of tainting for the popular JVM and .NET runtimes. While a great deal of research has gone into security of JavaScript within popular browsers, none of them currently support taint tracking. We feel that there is much to be done here to push these runtimes forward.

There is fact an interesting historical precedent for this kind of work influencing runtimes: research performed by Michael Ernst and his group [45] on type qualifier support for Java has eventually lead to Type Annotations language extension (JSR 308), which will be part of Java 8. He was the first non-Sun-employee to be the specification lead for a Java language change. This JSR was awarded "most innovative JSR" by Sun. Perhaps now is a good time for a similar effort for customizable runtime taint support.

What level instrumentation of instrumentation to use (see Figure 9)?
Is implicit tainting required?
What is the proper policy? Is the policy complete?
Does taint tracking need to be sound?
How to filter out the false positives? What kind of runtime overhead is acceptable?
Is static analysis possible for reducing the overhead?
What to do in the case of an observed taint violation?

Fig. 19. A list of questions to address before implementing a runtime taint tracking system.

IX. Conclusions

While one key goal is to provide a comprehensive overview of research on dynamic taint tracking in managed runtimes, we attempt to go beyond being merely a survey of related academic literature, by accomplishing the following goals. First, we aim to provide guidance for both researchers and practitioners implementing a runtime taint tracking system. To summarize the practical recommendations of this paper, Figure 19 shows a list of questions that someone needs to answer before trying to build a runtime tainting system. Second, we provide a formal foundation for most common forms of runtime taint tracking, captured in the form of an operational semantics. Third, we give a list of open problems solving which we hope will increase the level of adaption.

References

[1] Chris Anley. Advanced SQL injection in SQL Server applications. http://www.nextgenss.com/papers/advanced_sql_injection.pdf, 2002.

[2] Anonymous. StackShield. http://www.angelfire.com/sk/stackshield, 2002.

[3] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, 2009.

[4] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, pages 3:1–3:12, 2010.

[5] Dzintars Avots, Michael Dalton, Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the International Conference on Software Engineering*, May 2005.

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[7] Luciano Bello and Alejandro Russo. Towards a taint mode for cloud computing web applications. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2012.

[8] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. Waptec: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the Conference on Computer and Communications Security*, pages 575–586, 2011.

[9] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, volume 6961 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2011.

[10] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in Java. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 2–11, January 1999.

[11] CGI Security. The cross-site scripting FAQ. http://www.cgisecurity.net/articles/xss-faq.shtml.

[12] Erika Chin and David Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the ACM workshop on Secure web services*, SWS '09, pages 3–12, 2009.

[13] Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, pages 200–214, 2010.

[14] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[15] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, 2010.

[16] Fortify, Inc. Fortify SCA. http://www.fortifysoftware.com/products/sca/, 2006.

[17] Steve Friedl. SQL injection attacks by example. http://www.unixwiz.net/techtips/sql-injection.html, 2004.

[18] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference*, December 2005.

[19] William G. J. Halfond and Alessandro Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the International Conference on Automated Software Engineering*, pages 174–183, November 2005.

[20] William G. J. Halfond and Alessandro Orso. Preventing SQL injection attacks using AMNESIA. In *Proceedings of the International Conference on Software Engineering (formal demo track)*, May 2006.

[21] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 175–185, 2006.

[22] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[23] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, pages 13–26, May 1997.

[24] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 121–132, 2012.

[25] Emre Kıcıman and Benjamin Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of ACM Symposium on Operating Systems Principles*, October 2007.

[26] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, pages 199–209, 2009.

[27] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript instrumentation in practice. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, 2008.

[28] Dave King, Susmit Jha, Divya Muthukumaran, Trent Jaeger, Somesh Jha, and Sanjit A. Seshia. Automating security mediation placement. In *Proceedings of the European Conference on Programming Languages and Systems*, pages 327–344, 2010.

[29] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[30] David Litchfield. *SQL Server Security*. McGraw-Hill Osborne Media, 2003.

[31] Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, Stanford, California, 2006.

[32] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the Sympolisium on Principles of Programming Languages*, January 2013.

[33] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[34] Francesco Logozzo and Herman Venter. Rata: Rapid atomic type analysis by abstract interpretation – application to JavaScript optimization. In *Proceedings of the Conference on Compiler Construction*, 2010.

[35] Michael Martin and Monica S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium*, 2008.

[36] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security vulnerabilities using PQL: a program query language. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2005.

[37] Michael Martin, Benjamin Livshits, and Monica S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, October 2006.

[38] Microsoft Corporation. Microsoft web protection library. http:

//wpl.codeplex.com, 2012.

[39] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript, October 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[40] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium*, 2007.

[41] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, and David Evans. Automatically hardening web applications using precise tainting. Technical Report CS-2004-36, University of Virginia, 2004.

[42] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[43] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf, 2004.

[44] OWASP. OWASP enterprise security API. https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, 2012.

[45] Matthew M. Papi and Michael D. Ernst. Improving Java annotations to enable custom type qualifiers. http://pag.csail.mit.edu/javari/java-annotation-design.pdf, July 2006.

[46] RSnake. XSS cheat sheet for filter evasion. http://ha.ckers.org/xss.html.

[47] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[48] Mike Samuel, Prateek Saxena, and Dawn Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the Conference on Computer and Communications security*, 2011.

[49] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript.

[50] Prateek Saxena, Steve Hanna, Pongsin Poosankam, and Dawn Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS*, 2010.

[51] Prateek Saxena, David Molnar, and Benjamin Livshits. ScriptGard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the Conference on Computer and Communications Security*, October 2011.

[52] Ben Schmidt. Google analytics XSS vulnerability, 2011. http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/.

[53] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[54] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[55] Chris Shiflett. Foiling cross-site attacks. http://www.phparch.com/issuedata/articles/article_66.pdf, 2004.

[56] Yehuda Shiran and Tomer Shiran. *Learn Advanced JavaScript Programming*. Wordware, 1998.

[57] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar, and Nikolaj Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the Symposium on Principles of Programming Languages*, January 2012.

[58] Margus Veanes, Benjamin Livshits, and David Molnar. Fast and precise sanitizer analysis with BEK. In *Proceedings of the Usenix Security Symposium*, August 2011.

[59] Margus Veanes, David Molnar, Todd Mytkowicz, and Benjamin Livshits. Data-parallel string-manipulating programs. Technical Report MSR-TR-2012-72, Microsoft Research, July 2012.

[60] Wietse Venema. Taint support for PHP. https://wiki.php.net/rfc/taint, 2008.

[61] Larry Wall. Perl security and taint mode. http://perldoc.perl.org/perlsec.html.

[62] Wikipedia. Address space layout randomization. http://en.wikipedia.org/wiki/Address_space_layout_randomization, 2012.