Static Detection of Asymptotic Resource Side-Channel Vulnerabilities in Web Applications

Jia Chen Oswaldo Olivo Isil Dillig Calvin Lin
The University of Texas at Austin, United States
jchen@cs.utexas.edu oswaldo.l.olivo@gmail.com isil,lin@cs.utexas.edu

***Abstract*—Web applications can leak confidential user information due to the presence of unintended side-channel vulnerabilities in code. One particularly subtle class of side-channel vulnerabilities arises due to resource usage imbalances along different execution paths of a program. Such side-channel vulnerabilities are especially severe if the resource usage imbalance is *asymptotic*. This paper formalizes the notion of *asymptotic resource side-channels* and presents a lightweight static analysis algorithm for automatically detecting them. Based on these ideas, we have developed a tool called SCANNER that detects resource-related side-channel vulnerabilities in PHP applications. SCANNER has found 18 zero-day security vulnerabilities in 10 different web applications and reports only 2 false positives. The vulnerabilities uncovered by SCANNER can be exploited using cross-site search attacks to extract various kinds of confidential information, such as a user's medications or purchase history.**

I. INTRODUCTION

Web applications have become enormously popular due to the ubiquity of the Internet and the existence of rich development frameworks. Hence, in today's Internet-rich world, most people perform their daily activities, including banking and e-commerce, using web applications. Unfortunately, this growing popularity of privacy-sensitive applications has also led to a surge of illegal activities by hackers trying to steal confidential data.

To secure private data, web applications currently rely on a combination of network-level security mechanisms, such as encryption and firewalls, as well as application-level protection techniques (e.g., credential checks and session handling). While these mechanisms provide some degree of privacy assurance, they do not prevent the application from leaking confidential data through unintended communication channels, known as *side channels*.

Many side-channel leaks in web applications are related to resource usage (e.g., time or space). As an example, consider a health-related web application whose response time varies significantly, depending on whether the user is taking a certain medication. In this case, the server response time can reveal confidential information about the ailments of specific users. For instance, recent work [21] has shown that timing and response-size side-channel vulnerabilities can be exploited using *cross-site search attacks* in which the attacker submits a cross-site query with the user's credentials and observes the time it takes for the browser to respond to the query.

The large bulk of existing side-channel attacks (usually against codes that carry out cryptographical computation) exploit minor imbalances in resource usage [29], [22], [37]. Such attacks rely on the assumption that the attacker is able to precisely measure the time it takes to execute simple native operations like integer comparison on the target machine. However, despite recent advances in statistical measurement techniques [34], minor differences in resource usage are difficult to observe in *web-oriented settings*. On the other hand, if the attacker is given the ability to *amplify* the imbalance in resource usage, then the vulnerability becomes much easier to exploit. For example, consider an application whose response size is constant if the answer to a security-sensitive query is negative but linear in the size of the input otherwise. In this case, an attacker can observe a substantial difference in response size by supplying a sufficiently large input to the application. We refer to such vulnerabilities that exhibit an asymptotic difference in resource usage as *asymptotic resource side-channel* vulnerabilities.

In this paper, we introduce and formalize asymptotic resource side-channels, a class of vulnerabilities that can be exploited in settings where the attacker cannot precisely measure resource usage (e.g., web applications). To help developers safeguard their applications against such vulnerabilities, we also present a novel static analysis for automatically detecting asymptotic side channels. Our analysis employs a lightweight program abstraction that combines taint information with a coarse summary of resource usage. In particular, our analysis differentiates between variables that are untainted, secret-tainted, and user-tainted, and summarizes the resource usage of each code fragment as zero, constant, or unbounded. The key idea underlying our analysis is to identify branch conditions that are secret tainted and where the resource usage is unbounded along one branch but not the other. As we show experimentally, our proposed program abstraction is effective at detecting asymptotic side-channel vulnerabilities with a low false positive rate and can uncover exploitable vulnerabilities in real-world web applications.

We have implemented our proposed static analysis in a tool called SCANNER [1] for analyzing PHP applications. While the techniques we describe in this paper can be used to detect any resource-related asymptotic side-channel vulnerability, our implementation focuses on two kinds of resources, namely, *time* and *response size*. In addition to pinpointing vulnerable components, SCANNER further aids security analysts by iden-

[1] SCANNER stands for *Side Channel ANalyzER*

978-1-5386-2684-9/17/$31.00 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

tifying confidential database fields that may be leaked due to the detected vulnerability. Furthermore, SCANNER helps users assess the severity of the uncovered vulnerability by semi-automatically generating a Javascript exploit that performs a cross-site search attack.

We evaluate SCANNER on 10 open-source PHP applications and show that it uncovers 18 side-channel vulnerabilities and reports only 2 false positives. Furthermore, we are able to exploit these vulnerabilities using cross-site search attacks and show that the attacker can extract various kinds of confidential data, such as a user's purchase history, medical records, and bids placed by the user. We have reported the vulnerabilities to the developers and many of them have been fixed by the developers at the time of this submission.

II. FORMALIZATION OF ASYMPTOTIC RESOURCE SIDE-CHANNELS

In this section, we formally define the class of side-channel vulnerabilities studied in this paper and justify our decision to focus on this subclass.

Definition 1: **(Resource Side-Channel Vulnerabilities)** Let h, l denote *high* and *low* inputs of a program, respectively, and let $R_P(i)$ denote the resource usage of program P on input i. We say that program P has a *resource side-channel vulnerability* if:

$$\exists h_1, h_2, l.\ \ R_P(h_1, l) \neq R_P(h_2, l)$$

The above definition above is effectively an instantiation of the well-known *non-interference* policy [39] with respect to resource usage[2] As in traditional non-interference terminology, *high* variables represent secret values, while low variables denote values that are not security-sensitive. Hence, according to the above definition, a resource side-channel vulnerability arises if it is possible to observe different resource usages when program P is run on the same low input but different high inputs. Hence, an adversary can glean information about the secret simply by observing the program's resource usage.

Since Definition 1 does not specify the kind of resource, it is quite general and can be instantiated in a variety of ways to yield different classes of side-channel vulnerabilities previously discussed in the literature. For instance, if the resource of interest is CPU cycles, then this vulnerability corresponds to a *timing side channel*. On the other hand, if we instantiate R_P with power consumption, then the vulnerability could be exploited to cause a *power monitoring attack*.

However, since non-interference is a very strong condition that is violated by almost any program, we believe that such an approach is not practical. For instance, consider a program that has a *very minor* resource imbalance (e.g., a few CPU cycles) across two different execution paths. While such a program has a resource side-channel vulnerability according to Definition 1, it is very unlikely to be exploitable because an attacker cannot reliably observe this minor imbalance in resource usage. [3]

Motivated by the observation that standard non-interference is too strong of a policy to reason about *exploitable* resource side channels, we instead focus our attention on *asymptotic* resource side channels, which can be reliably exploited by attackers:

Definition 2: **(Asymptotic Resource Side-Channel Vulnerabilities)** Let h denote the high inputs of a program, and let $R_P(I)$ denote the resource usage of program P on input I. We say that program P has an **asymptotic resource side-channel vulnerability** if:

$$\exists h_1, h_2.\ \ R_P(h_1) \neq \Theta(R_P(h_2))$$

In this definition, observe that the low inputs are unconstrained, so $R_P(h_1)$ and $R_P(h_2)$ are both functions of low inputs l. Thus, this definition states that it is possible to find a pair of secrets h_1 and h_2 for which the resource usage of P will be asymptotically different with respect to the low-inputs. Since the attacker can control the program's low inputs, he can easily tell whether the secret is h_1 or h_2 by running the program on arbitrarily large values of the low input.

Observe that every asymptotic resource side-channel vulnerability also satisfies Definition 1, but not vice versa. We illustrate the differences between Definitions 1 and 2 using the following two examples.

Example 1: Consider the following code snippet:

```
foo(int n) {
   if (secret) {
       for(int i = 0; i < n; i++)  consume(1);
   } else consume(1);
}
```

Let $consume(x)$ be a statement that consumes x units of resource. If the value of `secret` is *true*, then the resource usage of `foo` is n. On the other hand, if `secret` is *false*, then the resource usage is 1. Since, $n \neq \Theta(1)$, this program contains an asymptotic side-channel vulnerability according to Definition 2.

Example 2: Consider the following code snippet:

```
bar(int n) {
   if(secret) consume(2); else  consume(1); }
```

Here, function `bar` contains a vulnerability according to Definition 1 because the resource usage of the program differs depending on whether `secret` is true or false. However, this function does not exhibit an *asymptotic* vulnerability because $R_P(H_1)$ and $R_P(H_2)$ only differ by a constant.

This second example illustrates why we choose to focus on the subclass of vulnerabilities given by Definition 2 as opposed to Definition 1: Because the difference in resource

[2]While the term "non-interference" was originally coined by Goguen and Meseguer [23], our definition follows the one adopted by the language-based security community [39], [8], [41].

[3]In fact, even if we restrict ourselves to resource-heavy operations, such as database queries, Definition 1 still seems to be too strong in practice. To justify this claim, we implemented an analysis that reports all instances of Definition 1 for resource-heavy operations (specifically, database queries). Even under this restrictive scenario, the analysis reported 285 warnings for the 10 PHP applications used in our evaluation. Our timing measurements for a subset of these reports reveal that the overwhelming majority of the warnings are not exploitable under realistic database states (because there is significant overlap between timing measurements for positive and negative queries).

usage is very small in Example 2, it is, in practice, quite hard to exploit this imbalance due to various kinds of noise in the program's execution environment (e.g., network traffic). Hence, our approach deliberately targets vulnerabilities that can be amplified by the attacker through carefully crafted inputs. In fact, previous work [21] has argued that one of the tools for exploiting side channel vulnerabilities is to *"amplify the side channel by inflating communication or computation"*. This trick only works if the program contains an *asymptotic* imbalance in resource usage rather than a constant difference.

Unfortunately, in practice it is still difficult to statically reason about worst-case resource usage, and existing techniques for reasoning about complexity typically do not scale to large programs [25], [24], [26], [12]. Hence, to analyze realistic web applications with a low false positive rate, we further restrict our attention to the following subclass of asymptotic side-channel vulnerabilities that can be detected using lightweight static analysis:

Definition 3: **(Constant-Superconstant Resource Side-Channel Vulnerabilities)** Let h, l denote the high and low inputs of a program, and let $R_P(I)$ represent the resource usage of program P on input I. Program P has a *constant-superconstant resource side-channel vulnerability* if:

$$\exists h_1, h_2.\ \ R_P(h_1) = O(1)\ \wedge\ R_P(h_2) \neq O(1)$$

In other words, we are interested in detecting a subclass of asymptotic side-channel vulnerabilities where the resource usage is constant for some values of the secret but a function of L for other values of the secret. Note that, in this definition, $R_P(h_2) \neq O(1)$ implies that the resource usage is a non-constant function of L when the high inputs are fixed to h_2. Our detection algorithm targets this specific subclass of vulnerabilities because it is possible to find instances of Definition 3 in a practical way without requiring heavy-weight resource usage analysis.

III. Static Detection of Asymptotic Resource Side Channel Vulnerabilities

We now turn our attention to the static analysis algorithm for detecting asymptotic (i.e., constant-superconstant) vulnerabilities. We first explain the key ideas underlying our algorithm and then provide a more formal description of the analysis.

A. Key Ideas

The key idea underlying our static analysis is to look for two different paths π_1, π_2 in the program such that π_1, π_2 satisfy the following conditions:

1) Paths π_1, π_2 differ on the value of the secret
2) Resource usage of π_1 is a function of low input l
3) Resource usage of π_2 is *not* dependent on l

To see how these conditions relate to Definition 3, observe that condition (1) partitions the high inputs into two disjoint classes H_1, H_2 such that $h_1 \in H_1$ satisfies the predicates on path π_1, while $h_2 \in H_2$ satisfies the predicates in path π_2. Furthermore, condition (2) states that $\forall h_2 \in H_2.\ R_P(h_2) \neq O(1)$, while condition (3) ensures $\forall h_1 \in H_1.\ R_P(h_1) = O(1)$.

$$\begin{array}{lcl}
\text{Expression } E & := & c \mid v \mid E_1\ \star\ E_2\ \ (\star \in \{+, -, ...\}) \\
\text{Condition } C & := & E_1 \circ E_2\ \ (\circ \in \{<, >, =\}) \\
 & & \mid \neg C \mid C_1\ \wedge\ C_2 \mid C_1\ \vee\ C_2 \\
\text{Statement } S & := & \mathit{consume}(E) \mid \mathit{source}(v, \mathit{label}) \\
 & & \mid v := E \mid S_1; S_2 \\
 & & \mid C\ ?\ \ S_1 :\ S_2 \mid \text{while } (C)\ \mathit{do}\ S
\end{array}$$

Fig. 1. Language used for describing our analysis

Our static analysis combines two kinds of taint information (namely, *secret taint* and *input taint*) with a coarse summary of resource usage for each program fragment. Input taint information is used to summarize the resource usage of each code snippet as zero, constant, or potentially infinite, where the last value indicates that resource usage is controlled by the attacker. In contrast, secret taint information is used to determine if two program paths differ with respect to some confidential data. Combining these two crucial pieces of information, our analysis reports an error if it encounters a secret-tainted condition such that the resource usage is potentially infinite along one branch, but not in the other.

Example 3: Consider again the code from Example 1, which contains an if statement whose branch condition is secret-tainted. The resource usage in the else branch is constant and does not depend on user input. On the other hand, the resource usage in the then branch can be made arbitrarily large because the loop bound `n` is *input-tainted*. Since the resource usage inside the loop body is non-zero and the loop bound is input-tainted, we summarize the resource usage of the then branch as potentially infinite. The analysis reports a potential vulnerability for this example because there is a secret tainted branch-condition, where the resource usage is potentially infinite along one branch, but not the other.

B. Formal Description of Static Analysis

We formally describe our algorithm on the simplified imperative programming language shown in Figure 1. In addition to the standard assignment ($v := E$), sequencing ($S_1; S_2$), conditional ($C\ ?\ S_1\ :\ S_2$), and looping constructs, this language also contains the following statements that model resource consumption and taint introduction:

- *Resource consumption:* The statement $\mathit{consume}(E)$ models the consumption of E units of resource, where E is an integer expression. For example, if the resource of interest is memory, then each memory allocation can be modeled using a consume statement in our language.
- *Taint introduction:* The statement $\mathit{source}(v, \mathit{label})$ models the tainting of variable v with label *label*. where *label* is either $\mathcal{H}$ (for high) or $\mathcal{L}$ (for low). For instance, $\mathit{source}(v, \mathcal{H})$ indicates that variable v is assigned to a secret value. In practice, taint sources with label H model database queries that retrieve security-sensitive data (e.g., password). On the other hand, taint sources with label $\mathcal{L}$ represent operations that accept some input from the user.

$$\frac{}{\Gamma \vdash c : \bot}$$

$$\frac{}{\Gamma \vdash v : \Gamma(v)}$$

$$\frac{\Gamma \vdash E_1 : \eta_1 \qquad \Gamma \vdash E_2 : \eta_2}{\Gamma \vdash E_1 \star E_2 : \eta_1 \sqcup \eta_2}$$

Fig. 2. Rules for determining taint values of expressions E

As mentioned earlier in Section III-A, our static analysis combines two different program abstractions to effectively detect asymptotic resource side channel vulnerabilities:

- *Taint abstraction:* Each variable has a taint value drawn from the domain $\mathcal{T} = \{\bot, \mathcal{L}, \mathcal{H}, \top\}$, where $\bot$ denotes lack of taint, $\mathcal{L}, \mathcal{H}$ represent input- and secret-taint respectively, and $\top$ represents *both* input and secret taint. We impose the partial order $\bot \sqsubseteq \mathcal{H}, \mathcal{L} \sqsubseteq \top$.
- *Resource abstraction:* We summarize the resource usage of each program fragment as a value drawn from the set $\mathcal{R} = \{0, 1, \infty\}$ where 0 indicates no resource usage, 1 indicates *constant* (but not necessarily unit) resource usage, and ∞ indicates that the resource usage cannot be statically bounded (i.e., because it is controlled by the user). We define a $\oplus$ operation on elements of set $\mathcal{R} = \{0, 1, \infty\}$ as follows:

$$\begin{aligned} &\forall x \in \mathcal{R}. \quad x \oplus \infty = \infty \\ &\forall x \in \mathcal{R}. \quad x \neq \infty \Rightarrow x \oplus 1 = 1 \\ &\forall x \in \mathcal{R}. \quad x = 0 \Rightarrow x \oplus 0 = 0 \end{aligned}$$

  We also define a total order $\succ$ on set $\mathcal{R}$ as $\infty \succ 1 \succ 0$. Finally, given $\Delta_1, \Delta_2 \in \mathcal{R}$, we write $\Delta_1 \gg \Delta_2$ if $\Delta_1 = \infty$ and $\Delta_1 \succ \Delta_2$.

Using these program abstractions, we can describe our analysis using rules of the form $\Gamma \vdash S : \Gamma', \Delta, \chi$ where:

- *Taint environments* Γ, Γ' map program variables to a taint value $\eta \in \{\bot, \mathcal{L}, \mathcal{H}, \top\}$;
- $\Delta \in \{0, 1, \infty\}$ summarizes resource usage of statement S;
- χ is a boolean value indicating whether or not a vulnerability is present in S

Hence, the meaning of the judgment $\Gamma \vdash S : \Gamma', \Delta, \chi$ is: *"If we execute statement S in an environment that satisfies Γ, then the resource usage of S is given by Δ and the taint environment after S is Γ'. Furthermore, if χ is true, then S may contain an asymptotic resource side channel vulnerability"*.

We describe our static analysis using the inference rules shown in Figures 2, 3, and 4. First, the helper rules from Figure 2 and Figure 3 allow us to determine the taint value η for each expression E and predicate C under taint environment Γ. According to the rules in Figure 2, constants are not tainted ($\bot$), and the taint value for each variable v is given by Γ. For composite expressions of the form $E_1 \star E_2$, the taint value is given by the join of the values of E_1 and E_2. For instance, if E_1 has taint $\mathcal{H}$ and E_2 has taint $\bot$, then the taint value if $E_1 + E_2$ is $\mathcal{H} \sqcup \bot = \mathcal{H}$. The rules for determining taint value of predicates are similar to those for expressions and are shown in Figure 3.

$$\frac{\Gamma \vdash E_1 : \eta_1 \qquad \Gamma \vdash E_2 : \eta_2}{\Gamma \vdash E_1 \circ E_2 : \eta_1 \sqcup \eta_2}$$

$$\frac{\Gamma \vdash C : \eta}{\Gamma \vdash \neg C : \eta}$$

$$\frac{\Gamma \vdash C_1 : \eta_1 \qquad \Gamma \vdash C_2 : \eta_2 \qquad \text{op} \in \{\wedge, \vee\}}{\Gamma \vdash C_1 \text{ op } C_2 : \eta_1 \sqcup \eta_2}$$

Fig. 3. Rules for determining taint values of predicates C

Let us now consider the main analysis rules presented in Figure 4. Here, rule (1) describes the analysis of a taint source of the form *source*$(v, \textit{label})$. In this case, the new taint environment Γ' is obtained from Γ by updating the taint value of v to *label*.

Rule (2) describes the analysis of *consume*(E) statements that model resource usage. Recall that resource usage is defined to be ∞ if expression E can be made arbitrarily large by an attacker. Hence, we first use the helper judgments from Figure 2 to determine the taint value η of E. If $\eta = \mathcal{L}$, then the resource usage of this statement is ∞ but constant otherwise.

Rule (3) describes taint propagation for assignments of the form $v := E$. As before, we use the helper rules from Figure 2 to determine the taint value η of E and update the taint environment by assigning v to η.

Rule (4) shows how we analyze sequence statements $S_1; S_2$. Observe that the resource usage of this statement is obtained by adding the resource usage Δ_1 of S_1 and Δ_2 of S_2 using the $\oplus$ operation defined earlier. Furthermore, $S_1; S_2$ contains a vulnerability if either S_1 or S_2 has a vulnerability; hence we take the disjunction of χ_1 and χ_2.

Rule (5) for conditionals is a bit more involved. Recall that $C \ ? \ S_1 : S_2$ exhibits a side-channel vulnerability if C depends on the secret and S_1 and S_2 have different resource usages. Hence, to determine if there is a vulnerability, we first check the taint value η of C using Figure 3. Clearly, if $\eta \not\sqsupseteq \mathcal{H}$, the statement does not introduce a vulnerability; hence $\chi = \textit{false}$ under this scenario. On the other hand, if C is secret-dependent (i.e., $\eta \sqsupseteq \mathcal{H}$), then an asymptotic side-channel vulnerability arises if the resource usage Δ_i is ∞ in one branch but constant or zero in the other branch. Hence, χ is *true* if $\Delta_1 \gg \Delta_2$ or $\Delta_2 \gg \Delta_1$, but it is false otherwise.

Continuing with rule (5), let us consider the taint values after analyzing $C \ ? \ S_1 : S_2$ under Γ. If the taint environment after S_i is given by Γ_i, then the value of each variable v after $C \ ? \ S_1 : S_2$ is given by $\Gamma_1(v) \sqcup \Gamma_2(v)$. Hence, the join operation $\Gamma_1 \sqcup \Gamma_2$ on taint environments takes the pairwise join for each variable.

Finally, let us consider the resource usage of the statement $C \ ? \ S_1 : S_2$, where the resource usage of each S_i is given by Δ_i. Since S_1 and S_2 cannot execute at the same time, the resource usage of $C \ ? \ S_1 : S_2$ is $\textit{max}(\Delta_1, \Delta_2)$, which is in fact the same as $\Delta_1 \oplus \Delta_2$.

The final rule in Figure 4 describes the analysis of loops. First, the assumption $\Gamma \vdash S : \Gamma, \Delta, \chi$ at the second line of Rule

$$(1)\quad \frac{\Gamma' = \Gamma[v \mapsto \mathit{label}]}{\Gamma \vdash \mathit{source}(v, \mathit{label}) : \Gamma', 0, \mathit{false}}$$

$$(2)\quad \frac{\Gamma \vdash E : \eta \qquad \Delta = \begin{cases} \infty & \text{if } \eta \sqsupseteq \mathcal{L} \\ 1 & \mathit{otherwise} \end{cases}}{\Gamma \vdash \mathit{consume}(E) : \Gamma, \Delta, \mathit{false}}$$

$$(3)\quad \frac{\Gamma \vdash E : \eta}{\Gamma \vdash v := E : \Gamma[v \mapsto \eta], 0, \mathit{false}}$$

$$(4)\quad \frac{\Gamma \vdash S_1 : \Gamma_1, \Delta_1, \chi_1 \qquad \Gamma_1 \vdash S_2 : \Gamma_2, \Delta_2, \chi_2}{\Gamma \vdash S_1; S_2 : \Gamma_2, \Delta_1 \oplus \Delta_2, \chi_1 \vee \chi_2}$$

$$(5)\quad \frac{\begin{array}{c}\Gamma \vdash C : \eta \qquad \Gamma \vdash S_1 : \Gamma_1, \Delta_1, \chi_1 \\ \Gamma \vdash S_2 : \Gamma_2, \Delta_2, \chi_2 \\ \chi = \begin{cases} (\Delta_i \gg \Delta_j) & \text{if } \eta \sqsupseteq \mathcal{H} \\ \mathit{false} & \mathit{otherwise} \end{cases}\end{array}}{\Gamma \vdash (C?\ S_1 :\ S_2) : \Gamma_1 \sqcup \Gamma_2, \Delta_1 \oplus \Delta_2, \chi_1 \vee \chi_2 \vee \chi}$$

$$(6)\quad \frac{\begin{array}{c}\Gamma \vdash C : \eta \qquad \Gamma \vdash S : \Gamma, \Delta, \chi \\ \Delta' = \begin{cases} \infty & \mathit{if}\ \Delta = \infty \vee (\Delta \succ 0 \wedge \eta \sqsupseteq \mathcal{L}) \\ 1 & \mathit{if}\ \Delta = 1\ \wedge \eta \not\sqsupseteq \mathcal{L} \\ 0 & \mathit{otherwise} \end{cases}\end{array}}{\Gamma \vdash \mathit{while}(C)\ \mathit{do}\ S : \Gamma, \Delta', \chi \vee (\eta \sqsupseteq \mathcal{H} \wedge \Delta' = \infty)}$$

Fig. 4. Rules describing our static analysis

(6) states that taint environment Γ is a fixed-point, hence, the taint environment after the loop is also Γ. Now, let us consider the resource usage of the loop $\mathit{while}(C)\ \mathit{do}\ S$. Clearly, if the resource usage of the body S is ∞ (resp. 0), then the resource usage of the loop is also ∞ (resp. 0). However, if the resource usage of S is constant (i.e., $\Delta = 1$), then the resource usage of the loop depends on the taint value η of predicate C. In particular, if $\eta \sqsupseteq \mathcal{L}$, then the number of loop executions can be controlled by the attacker, causing the resource usage to be statically unbounded. Hence, if $\Delta = 1$, resource usage Δ' of the loop is ∞ if $\eta \sqsupseteq \mathcal{L}$ but $\Delta' = 1$ otherwise.

The last issue to consider in the loop rule is whether there is a vulnerability. First, observe that the loop may have a vulnerability if the loop continuation condition C is secret-dependent. In particular, if C depends on a secret, the attacker might be able to learn the secret by observing the number of times the loop executes, which in turn can be inferred from the program's resource usage. To understand whether the loop introduces a vulnerability, observe that $\mathit{while}(C)\ \mathit{do}\ S$ can be rewritten as $C?\ (S; \mathit{while}(C)\ \mathit{do}\ S) :\ \mathrm{skip}$. Clearly, the resource usage of the else branch is 0, and since $\Delta' \succeq \Delta$, the resource usage of the then branch is precisely $\Delta \oplus \Delta' = \Delta'$. Thus, the loop has a vulnerability if $\eta \sqsupseteq \mathcal{H}$ and $\Delta' = \infty$.

IV. Design and Implementation of Scanner

A. Scanner Basics

Scanner analyzes PHP applications and detects two specific kinds of vulnerabilities involving *time* and *response-size*. Specifically, *timing side channel* vulnerabilities allow the attacker to infer confidential data by observing server response times. In contrast, *response-size side channels* enable the attacker to glean secret information by observing the size of the response.

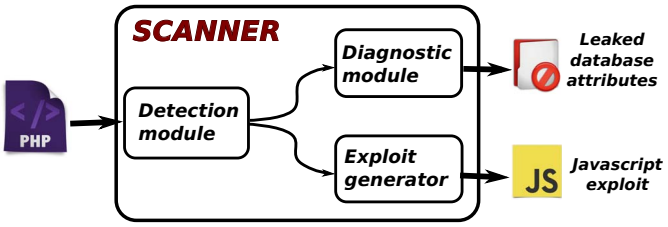

Fig. 5. Workflow of the Scanner tool

The workflow of the Scanner tool is shown in Figure 5. Internally, the Scanner tool consists of three different modules that perform complementary tasks:

- The *detection module* performs static analysis to flag potential asymptotic timing and response-size side-channel vulnerabilities. In essence, this module implements an instantiation of the algorithm described from Section III for two specific resource types.
- The *error diagnostic module* performs additional static analysis to report descriptive warnings. In particular, this module identifies confidential database fields that may be leaked by the application due to the uncovered vulnerabilities.
- The *exploit generation module* performs backwards symbolic execution to semi-automatically generate a Javascript program that can be used to exploit the uncovered vulnerability using a cross-site search attack.

Scanner is itself written in PHP and consists of ~5,000 lines of code. Our implementation uses the Z3 SMT solver [18] to solve the constraints collected during its symbolic execution phase for semi-automated exploit generation.

B. Detection Module

Scanner's detection module extends the static analysis described in Section III in two important ways: First, since PHP programs are not annotated with *source* and *consume* statements, we must reason about taint introduction and removal as well as resource consumption at the level of PHP statements. Second, our implementation is interprocedural and must deal with object-oriented features of the PHP language.

Annotations. To use Scanner, web developers need to annotate database fields that are considered to be confidential. In particular, rather than directly annotating the source code, Scanner users need to provide a separate file containing the security-sensitive attributes of each database table.

Taint sources. Our implementation considers two kinds of *taint sources*, namely, *user inputs* and *database operations*. User inputs are taint sources with label $\mathcal{L}$ and correspond to reads from pre-defined PHP arrays, such as `GET`, `POST`, and `SESSION`. In contrast, taint sources with label $\mathcal{H}$ correspond to database queries that retrieve private data from a database. Given a list of confidential database attributes, Scanner automatically infers whether or not a given `SELECT` statement is a taint source.

Sanitization. Recall that asymptotic resource side-channel attacks take advantage of the fact that the attacker can arbitrarily inflate resource usage by providing sufficiently large inputs. Hence, if the application *sanitizes* user input by bounding its size to a small value, then the attacker can no longer inflate resource usage. In our implementation, we consider checks that impose an upper bound on string length to be sanitizers. Similarly, string operations (e.g., `substr`) that limit the size of the resulting string are also considered to be sanitizers.

Resource consumption. In our implementation, *consume* statements from our formalization are instantiated in different ways depending on the type of side-channel vulnerability. For timing vulnerabilities, we consider every instruction to consume one unit of resource; hence, resource usage only becomes ∞ if a loop bound is tainted by user input. For response-size vulnerabilities, resource consumption corresponds to print statements. For instance, the resource usage of a statement `echo $foo` is ∞ if `foo` is tainted by user input, but has unit cost otherwise.

Interprocedural analysis. Even though we omit function calls from the formalization, our implementation is interprocedural and detects vulnerabilities that arise due to interactions between different functions. Specifically, our analysis computes *function summaries* of the form $(\Gamma, \Gamma', \Delta)$, where Γ is the input taint environment, Γ' is the output taint environment, and Δ represents resource usage of the procedure. The input and output taint environments used in the summary for f only mention memory locations that are accessible in f; hence, these summaries are re-usable. That is, if we analyze function f under the same input environment Γ, we can reuse its summary rather than re-analyzing the implementation of f.

C. Error Diagnostic Module

The static analysis described so far only allows SCANNER to detect the *existence* of a possible side-channel vulnerability. To help the user understand the severity and implications of the uncovered vulnerability, SCANNER performs an additional static analysis. Specifically, SCANNER's *error diagnostic module* outputs the set of confidential database attributes that may be leaked by the application. For instance, if SCANNER's output includes `Employees.Age`, this means that the attacker can infer something about the age an employee stored in the `Employees` database table.

To provide such diagnostic information, SCANNER performs a backwards static analysis that utilizes the information produced by the vulnerability detection module. Specifically, the input to the error diagnostic module is the *predicate of a conditional branch* along which there is a resource usage imbalance. Given such a predicate C, SCANNER then collects all secret-tainted variables used in C and performs backwards symbolic execution to trace each variable v to the database query that caused v to become tainted. Confidential database attributes that are mentioned in the `WHERE` clause of the query are then reported as being potentially leaked.

D. Exploit Generation Module

To further help programmers understand and assess the detected vulnerability, SCANNER also generates a Javascript program that can be used to launch a cross-site search attack to exploit the vulnerability. We first provide some relevant background on cross-site search attacks, and we then explain how SCANNER semi-automatically generates attack scripts.

Adversary model and XS Search attacks. Recent work has shown that *cross-site search attacks* can effectively exploit side-channel vulnerabilities in web applications [21]. XS-Search attacks require a very weak adversary model in which the attacker runs a malicious website but does not have man-in-the-middle or eavesdropping capabilities.

In this scenario, the attacker first tricks an unsuspecting user into executing a malicious script, for instance, by visiting the attacker's website or clicking on a link in a phishing email message. Now, the malicious script automatically submits a cross-site request with the user's legitimate credentials. Since web browsers allow a script to implement handlers for events triggered by cross-site requests, the malicious script can perform resource usage (e.g., timing) measurements between events and send this information back to the attacker. [4] Hence, if the underlying website contains a side-channel vulnerability, then the attacker can glean confidential information about the victim by inflating certain parameters used in a database query.

Backwards symbolic execution. The goal of SCANNER's exploit generation module is to (semi-automatically) synthesize Javascript programs that trigger the vulnerable component of the web application. In particular, because the vulnerable component may only be triggered under certain values of the URL parameters, our goal is to generate low inputs that exercise the vulnerable functionality.

Towards this goal, SCANNER starts from the vulnerable component C and performs backwards symbolic execution to collect all path constraints that are necessary for the execution to reach C. The output of the symbolic execution engine is an SMT formula ϕ such that a satisfying assignment to ϕ yields concrete values of the URL parameters that are sufficient to trigger the vulnerable functionality.

We emphasize that SCANNER's support for generating exploits is only *semi-automatic*: While our analysis infers the exact URL parameters needed to trigger the vulnerable functionality, the high inputs are merely placeholders that must be inflated by the user to amplify resource usage.

V. EVALUATION

We evaluate SCANNER by analyzing 10 widely-used open-source PHP applications. These applications include WordPress and Joomla (content management systems), OpenClinic (a medical records system), Gallery (a web-based photo album organizer), OpenCart, ZeusCart, and osCommerce (e-commerce), WeBid (an online auction software), and HotCRP and OpenConf (conference management systems). Some of these applications contain information that is clearly security-sensitive, such as medical records, account balances, and

[4] Observe that response-size side-channels can also be exploited using timing measurements since response parsing times are dependent on the response size.

TABLE I
SUMMARY OF OUR EXPERIMENTAL RESULTS.

Application	Lines of Code	Files	Functions	Number of Callsites	Timing Vulnerabilities	Response Size Vulnerabilities	False Positives	Lines of Annotations	Analysis Time (min:sec)
OpenConf	24,581	133	152	4,493	0	0	0	39	1:52
OpenClinic	30,849	180	526	6,766	0	2	0	27	1:09
WeBid	48,753	336	652	8,042	1	1	0	53	4:01
HotCRP	57,799	125	2,540	18,367	0	3	1	32	3:47
Joomla	59,820	2,563	14,259	124,894	0	2	0	22	5:11
Gallery	62,699	505	1,959	16,027	0	0	0	13	8:26
osCommerce	86,663	702	195	22,273	0	0	0	53	10:13
OpenCart	156,322	1,014	3,717	53,010	1	0	0	98	27:48
ZeusCart	166,400	612	8,798	23,354	5	0	0	44	31:06
Wordpress	298,643	577	5,866	60,717	0	3	1	15	49:57
Total	992,529	6,747	38,664	337,943	7	11	2	396	143:30

TABLE II
INFORMATION INFERRED BY THE ATTACKER

Application	ID	Information Leaked
OpenClinic	1	Does X have a medical record?
OpenClinic	2	Has X been prescribed medication Y?
WeBid	3	Does X have product Y on their watchlist?
WeBid	4	Is X bidding on product Y?
OpenCart	5	Has X bought downloadable product Y?
ZeusCart	6	Has X bought downloadable product Y?
ZeusCart	7	Does X have an order with amount Y?
ZeusCart	8	Has X bought anything between dates X and Y?
ZeusCart	9	Does X's order Y have processing status Z ?
ZeusCart	10	Is Y the account status for user X?
HotCRP	11	Is X an author of a submitted paper?
HotCRP	12	Is X the title of a submitted paper?
HotCRP	13	Is Z the abstract of a submitted paper?
Wordpress	14	Has X sent a private comment to the admin?
Wordpress	15	Is X the e-mail of a commentator in a post?
Wordpress	16	Is X the private comment sent to the admin?
Joomla	17	Has X authored a private article?
Joomla	18	Is X the title of a private article?

TABLE III
SUMMARY OF TIMING RESULTS FOR POSITIVE AND NEGATIVE QUERIES. TIMES ARE IN MILLISECONDS.

ID	Positive Query		Negative Query	
	Avg Time	Std Dev	Avg Time	Std Dev
1	1,523.71	64.85	2,020.32	137.60
2	1,655.69	29.84	2,358.24	119.24
3	96.27	10.23	513.35	22.67
4	2,593.63	113.80	1,225.34	18.47
5	265.13	16.21	29.27	4.44
6	329.26	14.68	32.61	8.85
7	4,241.29	821.32	630.82	80.96
8	4,580.50	121.90	668.72	62.65
9	4,616.64	130.10	676.79	99.84
10	3,740.37	468.63	798.55	64.92
11	1,612.79	204.89	343.49	167.43
12	1,758.41	93.46	271.01	17.87
13	2,000.76	318.38	317.93	64.06
14	839.20	56.08	502.07	43.62
15	801.15	15.35	483.67	64.28
16	796.30	37.07	467.50	30.85
17	1,172.84	243.14	486.17	27.92
18	1,090.41	135.98	488.25	47.94

purchase histories. Some of the applications we analyzed also provide well-documented privacy guarantees for information that *may* be considered security-sensitive. For example, Wordpress explicitly states that email addresses of users will not be disclosed.

We run our experiments on a server laptop with Ubuntu 14.04, a dual-core 2 GHz processor, 8 GB of RAM, and the Apache 2.4.18 web server, connected to a campus wireless network. The client is a desktop machine running Ubuntu 14.04, with a dual-core 3 GHz processor, 8 GB of RAM, and the Firefox browser (version 44.0).

A. Overview of Results

Table I gives statistics about the analyzed programs and summarizes the results of our evaluation. As shown in Table I, the analyzed programs are quite large and contain between 24K and 298K lines of code. SCANNER's running time on these applications is quite reasonable, with the largest application taking 49 minutes to analyze. Most importantly, SCANNER reports a total of 20 vulnerabilities, 18 of which are indeed exploitable. Among the vulnerabilities uncovered by SCANNER, 7 are timing side-channel vulnerabilities, and the rest are response-size side channels.

Table II summarizes the information that the attacker can learn by exploiting the underlying vulnerabilities uncovered by SCANNER. The leaked information is in the form of yes/no queries. However, we emphasize that the underlying vulnerabilities actually allow an attacker to infer lage parts of a database table by using these binary conditions as an oracle. For example, the vulnerability in OpenClinic can be exploited to infer all medications prescribed to a user. Similarly, an attacker can the use the vulnerability in OpenCart to learn all e-books purchased by a user or the vulnerability in WeBid to infer all bids of a user. Hence, these programs violate multi-run security [9].

B. Exploiting Vulnerabilities by Measuring Response Times

We now describe how the vulnerabilities uncovered by SCANNER can be exploited to infer the information listed in Table II. Since we do not assume that the attacker has man-in-

the-middle capabilities, we use cross-site search attacks [5] to exploit these vulnerabilities (although it may also be possible to exploit these vulnerabilities using other means). Note that according to our adversary model, the attacker is not able to directly measure the exact size of the server's response. Instead, he could perform time measurement through the victim's browser to indirectly infer the response size, since longer responses generally takes more time to load. As a result, *both time and response size side channel of a web application can be exploited by measuring server response times on the client side*.

Table III shows server response times when the answer to the queries listed in Table II is positive and negative, respectively. Since each of the vulnerabilities is caused by an asymptotic imbalance in resource usage, wecan amplify the difference between positive and negative queries by supplying artifically inflated search queries (e.g., a query containing a very large search string). The average and standard deviation metrics in this table correspond to the collection of 100 samples, with URL request sizes of at most 90K characters.

As we can see from the data in Table III, the response times vary significantly for positive and negative queries. For instance, OpenCart has an average response time of 265 ms if user X has bought product Y but 29 ms otherwise. Similarly, WeBid has a response time of 2594 ms vs. 1225 ms, depending on whether user X is bidding on product Y. Since response times vary significantly for all the queries from Table II, it is possible for an attacker to exploit the vulnerabilities uncovered by SCANNER.

In the next two subsections, we describe in detail two representative vulnerabilities uncovered by SCANNER.

C. Response-size Side-Channel in OpenClinic

Our first example is a response-size side-channel vulnerability found in OpenClinic, which is an open-source medical records system. This vulnerability reveals information about whether a patient has a medical record in the system.

Description of vulnerability. The OpenClinic application contains a medical record search page, which allows authorized users to search for medical records matching a given query string. The important point is that logical connectives are allowed to be included in the query string, and if the connective is set to "OR", then the database query will return all rows for which the patient name matches *any* of the keywords in the search string. If the table does not contain patients matching the query, then the application prints `No results found for X`, where X is the search string provided by the user. On the other hand, if there is a record matching the query, then the application displays the patient's record, whose length is independent of the search string. Therefore, one can learn whether the patient has a medical record or not by simply observing whether the response size is linear or constant with respect to the input string.

[5]Please see Section IV-D for background on cross-site search attacks.

Note that the attacker cannot directly see the patient's record since (a) he does not have access to the user's credentials, and (b) the same-original policy (SOP) prevents scripts contained in the attacker's website from directly accessing data in OpenClinic. However, since the attacker can perform time measurements in the victim's browser, he can observe the differences in time to infer the differences in response size.

Sample exploit. The attacker could inflate the query string by appending a long suffix (e.g., "aaaaaaaa...") after the patient of interest using the "OR" connective. If the patient name is in the database, the query will succeed and a short response will be returned. Otherwise, since the query string has been intentionally inflated by the attacker and also reflected by the server, the attacker can tell that a negative answer to the query will be associated with a much longer response length.

Possible fix. It is easy to fix the vulnerability in this code by not including the search string in the response message. For instance, the code would no longer be vulnerable if the server always responded "No results found for user" if the queried patient were not found. Alternatively, the code could sanitize the search keyword by ensuring that it does not contain more than a certain number of characters.

D. Timing Side-Channel in ZeusCart

Our second example is a timing side-channel vulnerability that SCANNER found in the ZeusCart e-commerce system.

Description of vulnerability. At a high level, ZeusCart allows a user with the right credentials to search order histories, which may include information like customer name, order ID, and the corresponding order status. If there is no matching purchase, the user quickly gets an empty results table *in constant time*. Otherwise, the application performs additional sanitization over the input whose running time is *linear in the size of the query string*. Since there is a significant difference in response times depending on whether a user has a specific order status, an attacker who does not have login credentials can infer the existence of specific purchases by specific users by performing timing measurements in the victim's browser.

Sample exploit. Since the query string is controlled by the user, the attacker can inflate one of the query parameters to amplify the running time. For instance, the attacker may append many white spaces to the user name parameter, and in cases where there is a matching purchase, the attacker can observe a significant increase in running time of the application. Specifically, as shown in Table III, positive queries take 3740ms on average, while negative queries take 799ms.

Possible fix. One possible fix for this vulnerability is to sanitize the query string by ensuring that it does not exceed a certain number of characters. This modification would ensure that the response time is bounded by a constant.

E. Threats to Validity

Timing Measurement. In our evaluation, we use a specific machine and network environment when conducting timing measurements for positive and negative queries. Of course,

these measurements could be substantially different if the exploits were executed on other platforms. However, because the resource usage imbalances detected by SCANNER are always *asymptotic*, it is possible for the attacker to further inflate the resource imbalance by providing even larger inputs.

Application Selection. Another threat to validity is that we evaluate our technique on 10 PHP applications; hence, our benchmarks may not be representative of many other real-world PHP applications. Nevertheless, we believe these applications constitute a good test-bed for our approach: Many of the projects we picked (e.g. Wordpress, Joomla) are well-known popular web applications with thousands of stars on Github, and their security has been widely audited by the open-source community.

VI. Possible Defenses against Asymptotic Resource Side Channel Vulnerabilities

The most straightforward way to eliminate the class of vulnerabilities discussed in the paper is to ensure that the attacker cannot inflate the resource usage imbalance between two execution paths that differ in the value of the secret. One simple way to achieve this goal is to sanitize user input by checking that its size cannot exceed some small, reasonable bound. Another (perhaps more robust) fix is to eliminate the asymptotic dependence on user input whenever possible. For instance, in the case of vulnerabilities that concern response size, a good rule of thumb is to make sure that the response does not contain the entire user input.

If the vulnerability occurs in a component that requires user or adminisrator credentials, another way to mitigate the vulnerability is to disallow cross-site read requests or employ anti-CSRF mechanisms. However, it is unrealistic to completely disallow cross-site read requests in many cases, and anti-CSRF defenses are typically only deployed against state-changing requests. Furthermore, even in the presence of anti-CSRF defenses, a response-size side channel still allows an attacker with traffic-monitoring capabilities to launch de-anonimization attacks and perform some sensitive data inference[43].

VII. Limitations of our Approach

SCANNER comes with a number of limitations that may result in both false positives as well as false negatives. First, the taint analyses described in Section III and implemented in SCANNER only propagate *explicit flows* (i.e. taint is only propagated due to assignments). Hence, our analysis can miss some resource side-channel vulnerabilities that arise due to implicit flows. For instance, consider the following function:

```
void foo(int input) {
  int c = 0; if (secret) c = 1;
  if(c) consume(input); else consume(1); }
```

This code has an asymptotic side-channel vulnerability since the resource usage is constant in one branch but user-controlled in the other one. However, SCANNER will not report this vulnerability because there is no *explicit flow* from `secret` to `c`. This limitation can be circumvented by tracking implicit flows, albeit at the risk of introducing more false positives.

Second, many of the vulnerabilities uncovered by SCANNER are exploited using XS search attacks (recall Section IV-D). However, if the application employs anti-CSRF mechanisms and the vulnerable component requires log-in credentials, then the reported vulnerabilities may not be exploitable using XS search attacks. In fact, the false positives reported in Table I are both caused by SCANNER's inability to reason about anti-CSRF measures.

Third, our analysis only detects a subclass of resource side-channel vulnerabilities where the resource usage is constant in one execution path but dependent on the user input in another execution path (recall Definition 3). However, it is nonetheless possible to exploit a constant, but very large, imbalance in resource usage. The techniques presented in this paper do not address such vulnerabilities.

VIII. Related Work

Static analysis for web security. There has been much previous work on static analysis for web security. Most of these techniques focus on automated detection of XSS and SQL injection vulnerabilities [31], [44], [46], [28], [4], [45], [33]. Dahse and Holz also consider *second-order* XSS and SQLi vulnerabilities and propose a static analysis for automatically finding them [17]. In constrast to these approaches, our work focuses on identifying privacy vulnerabilities caused by an observable resource usage imbalance in the program.

Side channels in web applications. While side-channel leaks have been known for decades, the first thorough study of side-channel leaks in web applications is presented by Chen et al [15]. Subsequent works include Chapman and Evans' black-box testing [14], which is a pure dynamic analysis, and Side-buster [48], which uses a hybrid of static and dynamic analysis. Our approach differs from them in several ways: First, while those tools detects privacy leaks caused by an imbalance in the number of requests exchanged between a client and the server, it does not reason about side-channel leaks originating from individual requests. Second, the vulnerabilities discovered by the aforementioned tools can only be exploited by an attacker who is able to sniff network traffic, which requires the attacker to be in the same network path as the victim. In contrast, our approach detects vulnerabilities that can be exploited remotely through cross-site search attacks.

Web timing attacks. Felten and Schneider present one of the first case studies on web timing attacks [20]. In later work, Brumley and Boneh show how to extract private keys from web servers running OpenSSL using timing attacks [11]. Bortz and Boneh describe web timing attacks in which they obtain valid usernames and items in users' shopping carts [10].

In more recent work, Gelernter and Herzberg introduce *cross-site search attacks* as a mechanism for exploiting side-channel vulnerabilities [21]. Van Goethem et al. show how to exploit multimedia tags in HTML5 to estimate response sizes during web timing attacks[42]. Recently, Van Goethem et al. have shown how a weakness in the Quota Management API in Javascript can be used to infer response sizes at the byte level

of granularity, allowing an attacker to de-anonimize Twitter users and obtain medical information from WebMD[43]. In this work, we use known attack techniques (specifically, [21], [42]) to assess the severity of the vulnerabilities uncovered by our approach. However, we emphasize that our work aims to detect application-specific vulnerabilities rather than describing a new class of web timing attacks.

Analysis for side-channel vulnerabilities. There has been significant research effort on verifying non-interference, including techniques based on *self-composition* [8], *product programs* [7], [41], and *Cartesian Hoare Logic* [40]. In contrast to these techniques, our approach focuses on asymptotic resource side-channel vulnerabilities and detects them using lightweight static analysis rather than heavy-weight verification techniques that require the inference of precise loop invariants.

Pasareanu et al. have recently proposed a symbolic execution technique for generating inputs that maximize differences in timing and memory usage between multiple program runs [37]. Their technique does not differentiate between constant and asymptotic differences, and their evaluation is performed on small programs (e.g., modular exponentiation) rather than large-scale web applications.

Another related approach is CacheAudit [19], which analyzes x86 binaries with a given cache configuration. CacheAudit computes an overapproximation of the number of bits leaked by the application on a simplified machine model. Their primary focus is hardware-level cache adversaries, whose capabilities do not align with those of application-level web adversaries. For instance, it is easier for web adversaries to exploit multi-run vulnerabilities [9], while it is harder for them to measure nanosecond-level timing differences.

Language-based information flow. There is a large body of work on language-based solutions for finding violations of the non-interference principle and quantifying information leak [3], [16], [35], [32]. However, these techniques do not detect privacy vulnerabilities due to convert channels, such as timing or response size.

Zhang et al. [47] propose a language-based approach for tracking side-channel leakage. Their approach requires the program to be written in their proposed language, whereas our goal is to analyze existing web applications. Barthe et al. [6] propose a type-based solution to verify the absense of side-channels, but their technique is tailored towards cryptographic implementations and simply rejects programs that do not execute in constant time.

Taint analysis. A common approach for tracking information flow in existing applications is to use taint analysis. Hence, there is a large body of work on taint analysis for languages like Java (e.g. FlowDroid [2]) and C++ (e.g. FlowTracker [38]). While our work leverages taint information, our main contribution is an algorithm for statically identifying asymptotic resource side channels. Our implementation does not leverage mature taint analysis tools like FlowDroid, as we target web applications written in PHP.

Multi-run security. The amount of information that is leaked due to a side channel vulnerability is often related to the adversary's ability to aggregate secret information across multiple executions. Köpf and Basin [30] develop an information-theoretical model that quantifies information leakage involving multiple adaptive interactions between the victim and the adversary. Birgisson and Sabelfeld [9] propose an alternative model based on knowledge sets and show how to enforce 1-bit multirun security. Pasareanu et al. [37], [5] use symbolic execution and model counting to compute single-run information leakage bounds and reason about k runs o by applying their single-run technique to the k-composition of the original program. While our analysis does not quantify leakage, the vulnerabilities uncovered by SCANNERin our evaluation can be exploited to infer significant chunks of a database (see Section V-A). We leave it to future work to develop analyses that can quantify information leakage through asymptotic side channels across multiple program runs.

Static analysis for resource usage. There is a significant body of work on static analysis for determing worst-case resource usage of programs [25], [24], [26], [12], [36], [1], [27], [13]. However, many of these analyses are quite heavyweight and do not address privacy implications of resource usage. Since the focus of our paper is detecting *asymptotic* resource side-channels, we can effectively detect security vulnerabilities using a lightweight program abstraction that combines taint information with a coarse summary of resource usage.

IX. Conclusions

In this paper, we introduced *asymptotic resource side channels* and described a static analysis for detecting them. Our method uses an effective, but lightweight program abstraction that combines taint information with a summary of resource usage. We have used our tool SCANNER to analyze 10 open-source PHP applications and found 18 exploitable security vulnerabilities. We believe that SCANNER can help web application developers by automatically uncovering easy-to-exploit vulnerabilities in their applications.

Acknowledgement. This material is based on research sponsored by DARPA award FA8750-15-2-0096 as well as NSF Award CCF-1712067. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

References

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini, "Cost analysis of java bytecode," in *European Symposium on Programming*. Springer, 2007, pp. 157–172.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014, pp. 259–269.

[3] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *Symposium on Security and Privacy*. IEEE Computer Society, 2009, pp. 141–153.

[4] 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 *Symposium on Security and Privacy*. IEEE, 2008, pp. 387–401.

[5] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. ACM, 2016, pp. 193–204.

[6] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014, pp. 1267–1279.

[7] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *FM 2011: Formal Methods*. Springer, 2011, pp. 200–214.

[8] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*. IEEE, 2004, pp. 100–114.

[9] A. Birgisson and A. Sabelfeld, "Multi-run security," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 372–391.

[10] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *World Wide Web*. ACM, 2007, pp. 621–628.

[11] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *USENIX Security Symposium*. USENIX Association, 2003.

[12] Q. Carbonneaux, J. Hoffmann, and Z. Shao, "Compositional certified resource bounds," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2015, pp. 467–478.

[13] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann, "Relational cost analysis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. ACM, 2017, pp. 316–329.

[14] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. ACM, 2011, pp. 263–274.

[15] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 191–206.

[16] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *J. Comput. Secur.*, vol. 15, no. 3, pp. 321–371, 2007.

[17] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *USENIX Security Symposium*. USENIX Association, 2014, pp. 989–1003.

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[19] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. USENIX, 2013, pp. 431–446.

[20] E. W. Felten and M. A. Schneider, "Timing attacks on web privacy," in *Computer and Communications Security*. ACM, 2000, pp. 25–32.

[21] N. Gelernter and A. Herzberg, "Cross-site search attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1394–1405.

[22] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs," in *Proceedings of the 16th Cryptographic Hardware and Embedded Systems - International Workshop*, 2014, pp. 242–260.

[23] J. Goguen and M. Jose, "Security policies and security models," in *Symposium on Security and Privacy*. IEEE Computer Society Press, 1982, pp. 11–20.

[24] S. Gulwani, "Speed: Symbolic complexity bound analysis," in *Computer Aided Verification*. Springer, 2009, pp. 51–62.

[25] S. Gulwani, K. K. Mehra, and T. Chilimbi, "Speed: precise and efficient static estimation of program computational complexity," in *ACM SIGPLAN Notices*, vol. 44, no. 1. ACM, 2009, pp. 127–139.

[26] J. Hoffmann, K. Aehlig, and M. Hofmann, "Multivariate amortized resource analysis," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 357–370.

[27] J. Hoffmann, A. Das, and S.-C. Weng, "Towards automatic resource bound analysis for ocaml," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. ACM, 2017, pp. 359–373.

[28] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 6–pp.

[29] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology*. Springer-Verlag, 1996, pp. 104–113.

[30] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 286–296.

[31] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *USENIX Security Symposium*, 2005.

[32] P. Malacaria, "Assessing security threats of looping constructs," in *Principles of Programming Languages*. ACM, 2007, pp. 225–235.

[33] M. C. Martin, V. B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *OOPSLA*. ACM, 2005, pp. 365–383.

[34] T. D. Morgan and J. W. Morgan, "Web timing attacks made practical," *Black Hat*, 2015.

[35] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," *Software release. Located at http://www. cs. cornell. edu/jif*, vol. 2005, 2001.

[36] J. Navas, M. Méndez-Lojo, and M. V. Hermenegildo, "User-definable resource usage bounds analysis for java bytecode," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 5, pp. 65–82, 2009.

[37] C. Pasareanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-smt," in *Computer Security Foundations Symposium*. IEEE, 2016.

[38] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016. ACM, 2016, pp. 110–120.

[39] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[40] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. ACM, 2016, pp. 57–69.

[41] T. Terauchi and A. Aiken, "Secure information flow as a safety problem," in *International Conference on Static Analysis*. Springer-Verlag, 2005, pp. 352–367.

[42] T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," in *Computer and Communications Security*. ACM, 2015, pp. 1382–1393.

[43] T. Van Goethem, M. Vanhoef, F. Piessens, and W. Joosen, "Request and conquer: Exposing cross-origin resource size," in *USENIX Security Symposium*. USENIX Association, 2016, pp. 447–462.

[44] G. Wasserman and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[45] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *PLDI*. ACM, 2007, pp. 32–41.

[46] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Usenix Security*, 2006.

[47] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. ACM, 2012, pp. 99–110.

[48] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "Sidebuster: Automated detection and quantification of side-channel leaks in web application development," in *Computer and Communications Security*. ACM, 2010, pp. 595–606.