Human-on-the-loop Automation for Detecting Software Side-Channel Vulnerabilities*

Ganesh Ram Santhanam[1], Benjamin Holland[1], Suresh Kothari[1], Nikhil Ranade[2]

[1]Iowa State University, Ames, Iowa 50011 | [2]Ensoft Corp., Ames, IA 50010
{gsanthan,bholland,kothari}@iastate.edu;nikhil@ensoftcorp.com

Abstract. Software side-channel vulnerabilities (SSCVs) allow an attacker to gather secrets by observing the differential in the time or space required for executing the program for different inputs. Detecting SSCVs is like searching for a needle in the haystack, not knowing what the needle looks like. Detecting SSCVs requires automation that supports systematic exploration to identify vulnerable code, formulation of plausible side-channel hypotheses, and gathering evidence to prove or refute each hypothesis. This paper describes human-on-the-loop automation to empower analysts to detect SSCVs. The proposed automation is founded on novel ideas for canonical side channel patterns, program artifact filters, and parameterized program graph models for efficient, accurate, and interactive program analyses. The detection process is exemplified through a case study. The paper also presents metrics that bring out the complexity of detecting SSCVs.

1 Introduction

Smartcards and satellite TV have been compromised by side channel attacks [21]. Hackers used a timing attack against a secret key stored in the Xbox360 CPU to forge an authenticator and load their own code [20]. Intelligence agencies have often relied on side-channel attacks to monitor their foes. Side-channel attacks have become a powerful threat to cryptography. One of the first papers on side channel attacks showed how to recover an RSA private key merely by timing how long it took to decrypt a message [18].

The possibilities are open-ended for the ways various program artifacts may be used to create side channels. Attackers exploit SSCVs by presenting a set of inputs and observing the space or time behaviors to construe the secret. The input could vary from HTTP requests to multiple log-in attempts depending on

* This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. 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 DARPA or the U.S. Government.

the application. The observable differential behaviors could be execution time, memory space, network traffic, or some output patterns of application-specific significance.

Since SSCVs are open-ended, the proposed approach is that of human-on-the-loop [10] automation. It incorporates: application-agnostic automation to determine hotspots, interactive automation to hypothesize SSCVs, and dynamic analysis for validating SSCVs. The automation uses static and dynamic analyses with program graphs as abstractions. We have developed tools that can handle Java source code or Java bytecode. The analyses are integrated and equipped with an interactive and compact visualization so that the analyst can experiment with and gain understanding of the program structure and behavior and apply that understanding to detect SSCVs. The SSCV toolbox is built using the Atlas platform [11].

The major research contributions are:

- A novel human-on-the-loop automation to detect SSCVs: It synthesizes a combination of non-trivial program analyses with program graphs as the enabling abstraction.
- Interactive automation: An interactive automation enabled by software visualization and querying capability to assist with hypothesizing and gathering evidence for side channels.
- Case study and complexity metrics: The case study brings out the significance of the proposed automation and how it can be used in practice. The metrics provide insights into the complexity of analyzing software for SSCVs and serve as a starting point for searching SSCV hot-spots in a given software.

Organization. The paper is organized as: Section 2 describes a motivating example, Section 3 describes the overarching SSCV patterns and a detection process, Section 4 describes fundamental challenges, Section 5 describes application-agnostic and application-specific automation, Section 6 describes parameterized graph models for interactive application-specific automation, Section 7 presents experimental results, Section 8 presents a case study of detecting an SSCV, Section 9 describes related work, and Section 10 concludes the paper.

2 Motivating Example

Consider a login application with valid user names as the *secret*, and the following side channel: by observing the result (login success or failure) of multiple login attempts and the corresponding response time, an attacker can deduce the secret.

As illustrated in Figure 1, the example has two control flow paths, one of which includes a loop to verify the password. Since the loop path is taken only if the user name is valid, observing the longer time it takes for this path creates

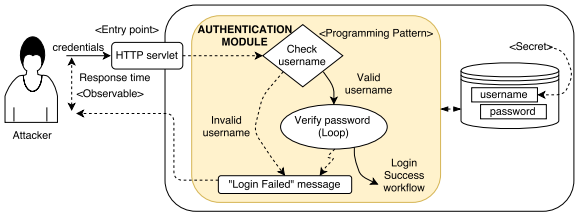

Fig. 1: Side channel in login application

the side channel. The programming pattern that creates the side channel in time includes: (a) the differential time on two control flow paths, (b) the loop that creates the differential, and (c) the branch condition (we will refer to it as differential branch) which governs the differential paths is tied to the secret. This example illustrates one of the overarching SSCV patterns proposed later (Section 3.2). This is an example of side channel in time. This example may also contain a side channel in space, if the application returns responses of different sizes to the client when the user name is valid and when the user name is invalid.

3 SSCV Detection Process

Detecting SSCVs is a nascent field of research. We are not aware of any literature that describes a systematic process for detecting SSCVs. We describe such a process as a starting point for developing the required automation.

Let us start by defining a three dimensional variability spectrum shown in Figure 2 corresponding to fundamental SSCV attributes: entry points, potential secrets, and programming constructs or artifacts causing differential behavior. Adversaries use *entry points* to provide inputs to induce differential behaviors, *secret types* are the broad categories of secrets that adversaries target, and *observables* are the space or time behaviors produced by program executions.

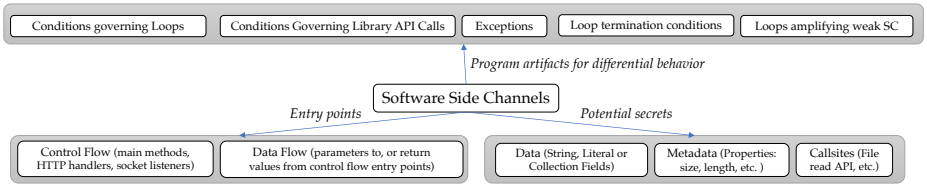

Fig. 2: Three variability dimensions of SSCVs

3.1 Human-on-the-loop Detection

Detecting SSCVs requires the following: (1) narrow down the possibility of SSCVs to a set of relevant program artifacts, (2) scrutinize these relevant artifacts to hypothesize plausible side channel exploits, and (3) employ dynamic analysis to confirm or reject the exploit hypotheses.

Phase I - Application-agnostic Analysis: Phase I is to pre-compute relevant program artifacts that serve as the foundation for the rest of the analysis. The challenge is to characterize relevant program artifacts. As discussed later, we have developed program artifact characterizations relevant to SSCVs. This automation is application-agnostic and it includes important program artifacts such as entry points, potential secrets, loops and branches, along with attributes relevant to SSCVs. For example, we use the DLI algorithm [29] to identify all loops in the bytecode, and characterize each loop based on attributes such as which library APIs it invokes (e.g., file, network, collection, etc.). The outcome of Phase I is a catalog of relevant program artifacts. The supporting application-agnostic automation is described in Section 5.2.

Phase I can reveal suspicious artifacts such as highly complex loops for operations that are often performed in practice by using library routines (e.g. a complex sorting loop). We call such artifacts smells and produce them as a part of Phase I.

Phase II - Application-specific Interactive Analysis: The challenge is to apply application-specific knowledge to develop hypotheses about SSCVs. The applications are often very large and the analyst is not expected to be intimately familiar with the code or the internals of the app. The analyst is expected to explore the app systematically to come up with SSCV hypotheses.

This is achieved in two steps:

1. Select a subset of program artifacts likely to lead to SSCVs.
2. Scrutinize selected program artifacts for the possibility of SSCVs to hypothesize how they could be exploited.

To facilitate the first step, we have automated filters based on the attributes in the program artifact catalog. The analyst can select program artifacts from the catalog that satisfy a combination of attributes. For example, if the analyst identifies that a collection in the app is the secret, he can select all loops (1) whose termination depends on the size of a collection, and (2) that perform file writes (contain callsites to file write APIs). The purpose of the filtering is to narrow down the likelihood of SSCVs to a small set of program artifacts.

To facilitate the second step, we have interactive automation using parameterized graph models, described in Section 6.

Phase III - Automatic Instrumentation: The challenge is to enable the analyst to validate or refute the hypotheses developed in Phase II. For this, the analyst needs to instrument relevant parts of the application, perform experiments to record observable events, and then analyze the results to conclude whether the hypothesized differential behavior is actually observable. The automation for this phase is not addressed in this paper. The techniques and tools for statically-informed dynamic analysis (SID) [17] can be used for Phase III.

3.2 Overarching SSCV Patterns

We propose overarching SSCV patterns as guideposts to analysts. We have designed the detection process and the supporting automation to facilitate analysis using these guideposts. The patterns are formulated around: (a) program artifacts that create space or time behaviors (loops or library calls), (b) program artifacts such as branches or exception handling mechanisms that create differential paths.

The five overarching patterns presented here are derived from our study of about 40 DARPA apps and also the reported crypto side-channel attacks [8,9,25] which amount to SSCVs. The example in Section 2 illustrates the first pattern.

1. *Differential behaviors caused by loops and governing branch condition*: The differential behavior is due to the presence of a loop in one path (as illustrated in the motivating example), or loops with different observable resource consumption in two different paths. In either case, the paths are governed by a branch condition predicated on the input and/or secret.

2. *Differential behaviors caused by Library APIs and governing branch condition*: The differential behavior here is due a branch condition predicated on the input/secret governing a path involving a call to a library API that can cause significant resource consumption (e.g., large array allocation) or produce some other distinct observable behavior (e.g., send a network packet or file I/O) that is absent in the other path governed by the branch condition.
3. *Differential behaviors caused by exceptions*: As exceptions trigger runtime control flow jumps, exception handlers for operations on the input/secret involving resource consuming loops or library calls can cause differential behavior.
4. *Differential behaviors caused by loop termination branch conditions*: When the secret is related to the number of iterations of a loop, the loop's termination branch condition itself serves as the governing branch condition for differential behavior. For example, if the size of a collection is the secret, a loop iterating the collection can consume time or memory proportional to the secret.
5. *Differential behaviors caused by weak side channels inside loops*: The differential behavior may be caused by a weak side channel, wherein the attacker may miss observables due to environmental noise. For example, in the second pattern above, a network packet sent by the app in one path (governed by a branch condition related to the secret) can be lost. However, if the governing branch condition is present within a loop, the weak side channel is amplified and can reveal the secret.

We have made available a repository [4] of example programs extracted from the vulnerable DARPA apps containing SSCVs related to the above patterns.

4 Fundamental Challenges of Detecting SSCV

Let us summarize the fundamental challenges that make detection of side channels quite difficult.

4.1 Path sensitive analysis

The challenge is to perform accurate analysis to account for individual behaviors along each of the control flow paths. The exponential growth of the number of paths makes the analysis difficult. The analysis must do the following: (a) account for the execution behavior along each CFG path, and (b) exclude the execution behavior along an infeasible path. Because of its high computational complexity, path-sensitive analysis is avoided in practice by aggregating the execution behaviors [2,1]. This aggregation is the major cause of the large number of false positives and negatives in static analyses.

4.2 Characterization of program artifacts

The challenge is to characterize relevant program artifacts that can cause SSCVs. Relevant program artifacts include application entry points, potential secrets, and control flow constructs in the code such as loops, branches and exceptions.

It is important to characterize relevant program artifacts and their specific attributes that define how the artifacts relate to the secret, how they create observable space/time behaviors, or how they create differential behaviors.

4.3 Incorporating application-specific knowledge

The challenge is to develop analyses that can account for the variability of application-specific notions of secrets and how they are revealed. The program artifacts obtained through application-agnostic analysis can be too many and they have to be narrowed down to correspond to secrets and side channel mechanisms that are application-specific.

5 Human-on-the-Loop Automation

Figure 3 gives an overview of the automation we have designed for detecting SSCVs. We employ static analysis as a funneling process to narrow down the set of program artifacts. The funneling process incorporates application-agnostic automation and application-specific automation. These correspond to Phase I and II of the detection process described earlier.

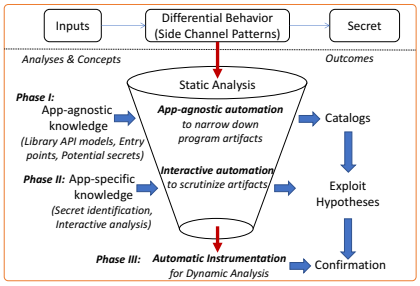

Fig. 3: SSCV Detection Overview

5.1 Relevant Program Artifacts

In this subsection, we discuss the specific analyses to characterize program artifacts with respect to the variability dimensions discussed in Section 3.

Entry Points: Application entry points are computed as starting points for auditing an application for SSCVs. Entry points identify the interfaces through which an attacker can supply input to induce differential behavior in the application. They also identify the part of the code that is reachable, thus helping the analyst to restrict his search for SSCVs to a subset of program artifacts.

Entry points are relevant with respect to the control flow and data flow. There are three kinds of control flow entry points corresponding to whether the application is standalone, web application, or peer-to-peer. These are Java main methods, HTTP request handlers defined in the application, and socket listeners respectively. Analogously for the data flow aspect, the parameters of the Java main methods, the parameters of the HTTP request handlers, the return values of the callsites that read from the sockets are of interest because it is through these interfaces that the attacker can provide inputs to induce differential behavior in the application. The possibility of SSCVs can be restricted to the subset of the artifacts tainted by the data flow entry points. For example, if the analyst suspects that a web application has a side channel pattern involving a loop (i.e., overarching SSCV patterns 1, 4, or 5), then it would suffice to audit only loops that receive data flow from the parameters to the HTTP request handler, namely the `javax.servlet.http.HttpServletRequest` objects.

Potential Secrets: An application can have several side channels but only side channels that reveal a secret are of relevance to the analyst. Identification of the secret in an application is therefore a critical step for the analyst to prune the search for SSCVs. It is one of the first steps the analyst performs using application-specific knowledge.

Secrets can be classified as *data* or *metadata*. Data secrets usually represent contents of variables. Examples of data secrets include primitive values (e.g., integer representing the maximum number of allowed login attempts with invalid password) and Strings (e.g., password or email address of administrator) defined in the app, as well as contents of collections. Examples of metadata secrets include the size of a String (e.g., the number of characters in the username or password) or a collection (e.g., the number of nodes or edges in a graph). Secrets can alternatively be classified as *simple* or *complex*. Secrets that can be deduced by knowing the contents of a single variable in the application are *simple* secrets. Secrets that are a function of multiple variables in the application are *complex*. An example of a complex secret is whether a graph is strongly connected or not. The secret here is a function of the nodes and edges in a graph that may be individual collections. Secrets may not always be present in the code. They can also reside in configuration files read by the application. In such cases, all returns from callsites to the file read APIs in the JDK and the libraries used by the application can be considered potential secrets. For example, in a login authentication application, if the set of valid usernames is the secret and it is read from a file or database, then artifacts corresponding to the returns of callsites that read from files or the database query result are the potential secrets.

Differential Behavior Artifacts: Analysis of relevant loops and branch conditions is key to detecting the presence of SSCV patterns in an application. A loop or branch is relevant to detecting an SSCV pattern if its behavior depends on the secret, i.e., *predicated on the secret*. For example, in the login authentication example, the loop used to verify the password is executed if the branch predicated on the username evaluates to true, and not otherwise. Therefore, the branch governing the loop for verifying password is relevant as it is predicated on the secret (username). A loop or branch is also relevant to detecting an SSCV pattern if it governs events that can be observed by the attacker such as file I/O, network operations, etc. Such a loop or branch is also relevant as they identify events that enable differential behavior to be observed by the attacker. Therefore, two kinds of loops are important to reason about SSCV patterns in an application: (a) Loops whose termination branch conditions are predicated on secrets (SSCV Pattern 4); (b) Loops whose body contains callsites to certain library APIs (SSCV Pattern 1,4,5). Similarly, two kinds of branch conditions are important: (a) Branch conditions predicated on secrets (SSCV Pattern 1,2,5); (b) Branch conditions governing certain library API calls (SSCV Pattern 2).

5.2 Application-agnostic Automation

This automation uses techniques based on classical static analyses (control flow, data flow, loop detection, taint analyses, etc.) to generate three catalogs, one each for: entry points in the application, potential secrets, and programming

constructs causing differential behavior in the application. The catalogs include all entry points, potential secrets, and loops and branches relevant to detecting SSCV patterns in the application as classified in the preceding subsections. The loop and branch catalogs identify loops and branches that specifically govern events related to file I/O and network operations, which are common observables in the applications we have encountered as part of our empirical study (Section 7). This can however be extended to include loops and branches that govern other events.

The catalog is saved within the Atlas platform as attributes of the program artifacts in Atlas. The branches and loops that match an SSCV pattern are tagged so the analyst can quickly select results from the catalog that match certain tags. The analyst can also export this information as a CSV file to facilitate spreadsheet-style filtering of the artifacts based on their attributes.

5.3 Application-specific Interactive Automation

We propose two broad categories for interactive analysis: (a) applying selective filters to narrow down the program artifacts, (b) using graph models to detect differential behaviors. The use of interactive analysis is brought out later through the case study presented in Section 8.

Selective Filtering of Program Artifacts: The filters are designed to narrow down the loops or the APIs that produce the observable space or time behaviors. The filters place one or more constraints to select a subset of program artifacts.

In order to enable filtering, each loop in the application is characterized with respect to the following properties: (1) nesting depth within the method containing the loop; (2) whether the loop's termination matches a common programming pattern (e.g., loop iterates over a collection using the `Iterator`'s `hasNext` API); (3) whether the loop is monotonic (variables controlling the loop's termination are either exclusively incremented or exclusively decremented within the loop); (4) categories of APIs (also called *subsystems*) invoked within the loop's body that indicate the kinds of observable events emitted by the loop's execution (e.g., network operations, file I/O, randomization, etc.); (5) the sizes of control flow and data flow graphs (numbers of nodes and edges) for the loop's body; (6) the number of callsites in the loop; (7) the number of paths within the loop that contain and do not contain callsites. The analyst can create custom filters with constraints on any subset of the above attributes to narrow down loops. We describe one of the commonly applied filters, namely *subsystem interactions filter* that selects loops based on the categories of APIs they invoke.

Subsystem Interactions Filter (SIF): The purpose of this filter is to enable analysts to narrow down the set of loops to loops that interact with a set of APIs. SIF has two configurable parameters: (a) an initial set of loops, and (b) a set of APIs. The analyst configures these parameters and invokes SIF. SIF selects loops that invoke the selected APIs directly or indirectly via function calls.

The resulting loops can be the ones that produce observable behaviors relevant to SSCVs. For example, if the analyst knows that the attacker could observe updates to the log files, he can configure SIF with the logging APIs to get loops

which include logging. The case study in Section 8 uses SIF to narrow down to loops involving network operations as a crucial step towards detecting an SSCV.

6 Interactive Automation: Parameterized Graph Models

Interactive automation is critically important for human-on-the-loop detection of SSCVs. The analyst must improvise and customize the analysis because of the open-ended possibilities for SSCVs. To be effective at it, the analyst needs to explore the software and gain insights with the help of interactive automation. We employ parameterized graph models as powerful abstractions for interactive automation. The generically defined graph models such as the call graph or the control flow graph are not customizable to solve specific problems. By not being able to focus on only the semantics relevant for a problem, their size explodes and they are no longer effective models to gain insights from. We employ novel graph models that can be parameterized to be problem-specific. The goal is to provide compact representations of software behaviors and structures relevant to SSCVs. These graph models have been developed through our ongoing research on graph models tailored to specific classes of cybersecurity and software safety problems [27,17].

Table 1 lists our current graph models, their input parameters, their outputs, and how they help the analyst in Phase II of the SSCV detection process to either *select* (narrow down) program artifacts, or to *scrutinize* narrowed down artifacts to hypothesize the presence of SSCV Patterns (Section 3.2).

Table 1: Interactive graph models and their use in Phase II of SSCV detection

Parameterized Graph Models	Input	Output	Phase II activity supported
Loop Call Graph	Method	Reachable methods containing loops	Select loops w.r.t. entry point
Loop Reachability	Dataflow artifact	Loops reached via dataflow	Select loops w.r.t secret
Taint	Source, Sink (dataflow)	Taint graph from source to sink	Select loops, branches w.r.t secret
Subsystem Interaction	Loops/Methods, APIs	Loops interacting with APIs	Select loops, methods w.r.t APIs
Projected Control	Method, Events	Reachability preserving CFG reduction	Scrutinize SSCV Pattern 1, 2, 5
Exceptional Control flow	Control flow artifact	CFG with exceptional flow semantics	Scrutinize SSCV Pattern 3
Termination Dependence	Loop header	Slice w.r.t. loop's termination condition	Scrutinize SSCV Pattern 4

We describe interactive automation using two examples of parameterized graph models: *projected control graph* (PCG) for intra-procedural exploration and the *loop call graph* (LCG) for inter-procedural exploration.

6.1 Intra-procedural Interactive Automation

This subsection describes an intra-procedural interactive automation using the *projected control graph* (PCG) [27] as a parameterized model. As noted in Section 4, path sensitive analysis is a fundamental challenge for detecting SSCVs. SSCVs are rooted in differential behaviors. Analyzing each path is computationally impractical because the number of control flow paths grows exponentially as 2^n for n non-nested 2-way branch nodes. The PCG provides an efficient and accurate model to counter the path explosion issue by focusing on problem-specific distinct behaviors as opposed to distinct paths. As brought out by the study [27], it is an effective solution because the number of problem-specific behaviors do not grow exponentially.

The PCG is parameterized by the set of events relevant to a problem. Events correspond to the nodes of the control flow graph. The behavior along a control flow path is the *event trace*, the sequence of events along that path. The paper [27] formally defines the *event trace*, and an efficient algorithm to compute the PCG.

To create the PCG, two interactive modes are desirable in practice: (a) the analyst selects the events interactively, or (b) the analyst invokes an automated analyzer to create the set of events. We exemplify the two modes. Instead of SSCVs, we use a relatively simple example of *division-by-zero* vulnerability to bring out the gist of interactive automation with the PCG.

In the first interactive mode, the analyst selects the events interactively by clicking on the control flow graph (CFG). In the second interactive mode, the analyst invokes an analyzer to gather the relevant events. For this vulnerability, the analyst invokes the *backward slice analyzer*. Let us describe the first mode in detail. Figure 4 shows a code snippet with an instance of *division-by-zero* vulnerability on line 23. The four selected nodes are tick marked, they correspond to code lines 5, 17, 21, and 23. Instead of clicking on the CFG nodes, the analyst could also click on these lines of code.

Figure 4 shows how the PCG gets refined as the analyst selects the events by clicking on the CFG:

Interaction 1: Analyst clicks on two CFG nodes for statements 17 and 23. That generates the first PCG with two branch nodes and two marked event nodes (colored yellow).

Interaction 2: Analyst clicks on the third CFG node for statement 5. It generates the second PCG as a refinement.

Interaction 3: Analyst clicks on the fourth CFG node corresponding to statement 21. It generates the final PCG for the given vulnerability.

Let us now discuss the significance of the final PCG. Unlike 6 paths in the CFG, the PCG has only 3 paths corresponding to 3 distinct behaviors relevant to the given vulnerability. The relevant behaviors (event traces) are: (a) 5,17,23; (b) 5,23; and (c) 5,21,23. The behavior (a) leads to the *division-by-zero* vulnerability. The governing branch condition for these two paths is $c2 = TRUE$ and $c3 = FALSE$. Note that the *path feasibility* analysis is simplified. The governing branch condition must be satisfied for the vulnerability to occur.

PCG use case in detecting SSCVs: Let us describe an use case of PCG for detecting SSCVs. Suppose that an analyst has selected a loop L for scrutiny. Specifically, the analyst would like to know which branch conditions in the application govern the execution of L. This requires the analyst to examine control flow paths from the entry point of the application to L. Using the CFG for this task would involve aggregating the CFGs over all methods in the call graph from the entry point to the method containing L. As shown later using an empirical study in Section 7, such inter-procedural CFG are typically very large and prohibitively expensive to analyze. Instead, the analyst can construct a PCG starting with entry point method and prescribing the loop L as the relevant event. The resulting PCGs would retain only the relevant paths from the entry point that lead to L, and elide all other inter-procedural control flow. The

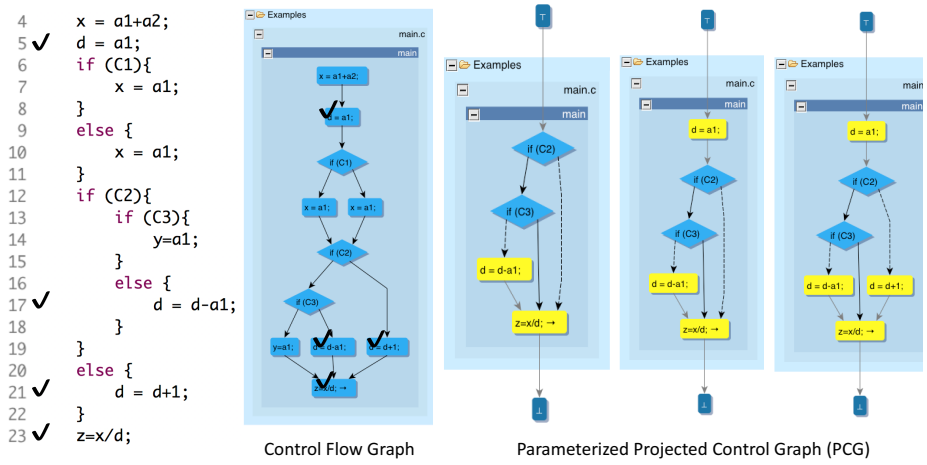

Fig. 4: Interactive automation with Projected Control Graph

graph size reduction from CFGs to PCGs for this use case is a good metric of the PCG's usefulness for SSCV detection. We show by experiments in Section 7 that the graph size reduction from CFGs to PCGs is significant.

6.2 Inter-procedural Interactive Automation

This subsection describes an inter-procedural interactive automation using the *Loop Call Graph* (LCG) [17] as a parameterized model. The LCG is a subgraph of the call graph that succinctly represents how loops are distributed across methods in the application. The LCG contains two types of nodes: (a) methods containing loops and (b) methods from which other methods containing loops can be reached via the call graph. A node is colored blue if it contains loops, and grey otherwise. There is an edge from method m_1 to method m_2 in the LCG if m_1 calls m_2. The LCG captures two important pieces of information crucial to analyzing side channels.

Inter-procedural Nesting: First, the LCG captures information about how loops are inter-procedurally nested by using visual (color) attributes for edges. An edge is colored yellow if the callsite of m_2 is located within a loop in m_1 (and black otherwise). This indicates that loops in m_2 are inter-procedurally nested within a loop in m_1. For example, consider a function f containing two loops $L1$ and $L2$, $L2$ nested within $L1$. This can be refactored into two methods f, g, with f containing $L1$, g containing $L2$, and f calling g from inside $L1$. A simple loop detection algorithm would not detect that $L2$ is actually nested within $L1$ although they are not in the same function. The LCG captures this information via a yellow edge from f to g. Moreover, the loop $L2$ may be further down in the call graph – such as in a function h called by g. The analyst can directly use the LCG to infer this inter-procedural nesting. The analyst can infer the presence of a recursive call; the LCG shows it as a cycle with a yellow edge.

Reachability: Second, the LCG is useful to understand how a particular loop can be reached from a selected entry point. An analyst can use this information to restrict the scope of SSCVs to the subset of the application reached by the LCG. For example, a web application could have several HTTP Servlets (Java server side component for handling HTTP requests). To find whether the application has a side channel by which an attacker can deduce a valid username, it may suffice for the analyst to analyze only loops reachable from the Authentication Servlet. The LCG with the Authentication Servlet's HTTP handler method selected shows only methods containing loops reachable from this entry point.

7 Empirical Evaluation with Challenge Apps

We study how the application-agnostic and application-specific automation described in Sections 5.2 and 5.3 help address the three fundamental challenges (Section 4). We present experimental results and evaluation of our tooling with respect to the first two challenges. We illustrate how our process and tooling help address the third challenge using a case study in Section 8.

Data Set. This empirical study covers a total of 40 challenge apps provided by DARPA, which have been specifically engineered to include software side channel vulnerabilities involving a broad range of entry points, secrets and programming constructs covering the overarching SSCV patterns discussed in Section 3.2. Overall, the data set includes 7,788 loops and 44,542 branch conditions.

7.1 Application-agnostic Characterization of Program Artifacts: Usefulness of Catalogs

Table 2 shows the distribution of loops across apps. The rows group the apps based on the number of loops they contain (buckets increment by 100 loops). The column *L1* is the average fraction of loops (across apps in each row) whose termination is predicated on a potential secret (the length of a `String` or size of a `Collection`). The column *L2* is the average fraction of loops (across apps in each row) that govern observable events (file I/O or network operations).

Table 2: Distribution of loops relevant to SSCV patterns

# Loops	# Apps	L1	L2
0-100	10	94.3%	14.5%
100-200	20	97.2%	14.3%
200-300	5	90.6%	8.2%
> 300	5	98.7%	5.3%

That a large fraction of loops is identified by *L1* shows that without using application-specific knowledge about the secret, there may be too many loops to scrutinize for the possibility of SSCVs. Therefore, identification of secret is a key first step that the analyst should perform as part of application-specific reasoning in Phase II. In contrast, the fraction of loops identified by *L2* narrows the search for the possibility of SSCVs effectively even without applying application-specific knowledge. Note that the current *L2* fraction shown in Table 2 only considers loops governing file I/O and network operations, whereas for certain applications there may be additional events of interest such as call to a specific library (which may in turn emit observable events) or recursion (which may be observable through the growth of stack size). Nevertheless, such special events can be

expected to be limited to a small subset of loops, and the loops identified by *L2* may be used by the analyst to proceed to Phase II.

Table 3 shows the distribution of branch conditions across apps. The rows group the apps based on the number of branches they contain. *B1* is the average fraction of branches predicated on a potential secret (the length of a `String` or size of a `Collection`). *B2* is the average fraction of branches that govern observable events (file I/O or network operations). Both *B1* and *B2* identify a small fraction of branches in the applications as relevant to detection of SSCVs This shows the usefulness of the branch catalog generated in Phase I - that the possibility of SSCVs due to branches can be narrowed down significantly even prior to applying application-specific knowledge.

Table 3: Distribution of branches relevant to SSCV patterns

# Branches	# Apps	B1	B2
0-500	17	3.2%	17.0%
500-1000	5	4.0%	10.6%
1000-2000	12	1.7%	6.9%
$>$ 2000	6	2.7%	5.2%

7.2 Usefulness of Parameterized Graph Models

Once the analyst has identified a set of loops or branches in the application using the catalog, in Phase II his objective is to develop a hypothesis about how these program artifacts may be exploited to reveal the secret. For this, the analyst has to understand whether the set of paths they induce cause observable differential behavior. Section 6 described two parameterized graph models: LCG and PCG, and how they can be composed to help the analyst understand the equivalence classes of paths from the entry points to a given loop L in the program.

We perform the following experiment using the graph models. For each loop L, we compute the LCG from the entry points to the method containing the loop L, and measure the aggregated number of nodes and edges in the CFGs of all methods in that LCG. We also construct PCGs with entry point and loop L as event, as described in the PCG use case for detecting SSCVs (Section 6.1).

Figure 5 shows the comparison of the number of nodes in the CFG versus the PCG (a, b respectively) and the number of edges in the CFG versus the PCG (c, d respectively). The results are presented as histograms with three bins designed to represent small, medium and large instances of the graphs (CFG or PCG). The bin sizes are set to 60 for nodes and 200 for edges, so that the small, medium and large instances correspond to easy, moderate and hard instances for an analyst to comprehend the instance (reachability of the loop from entry point). For example, a loop whose reacha-

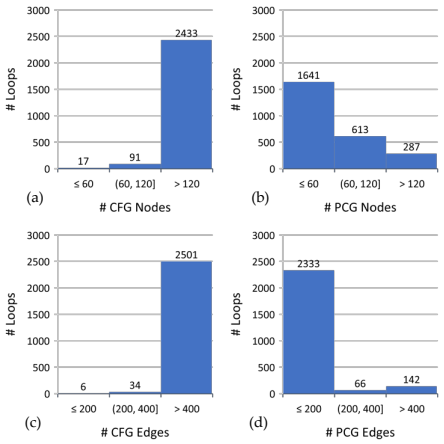

Fig. 5: Distribution of loops w.r.t. size of the CFG and PCG from the entry points to loops

bility from entry point according to this experiment involves 50 nodes and 150 edges is considered relatively easy. The set of inter-procedurally reachable paths was prohibitively expensive to compute for all the loops. Therefore, the histograms show results for 2542 loops from our data set.

Our results show that, compared to the CFG, the PCG significantly reduces the graph sizes while retaining the SSCVs relevant behaviors. The number of nodes and edges in the PCG was only 13.5% and 2.57% of those in the CFG, which is a drastic reduction. Furthermore, we uniformly observe across the histograms that when using the CFGs, most instances are hard (last bin) and very few are easy (first bin). Whereas the use of PCG reverses the trend: most instances fall in the easy (first) bin, and very few fall in the hard (third) bin.

8 Case Study: SSCV Detection

This section is a study using a challenge application from the DARPA STAC program.

Challenge application. The Law Enforcement Employment Database Server (LEEDS) is a network service application that provides access to records about law enforcement personnel. Employee information is referenced with a unique employee ID number. The database contains restricted and unrestricted employee information. The ID numbers of law enforcement personnel working on clandestine activities is restricted information. The database supports the following functionality: (1) *Search* - search law enforcement personnel by a range of employee ID numbers; (2) *Insert* - create a new employee ID number. Users can search, view, add unrestricted employee IDs and associated information. If a user makes a query for a range of IDs that contains one or more restricted IDs, the restricted IDs will not be included in the returned data.

Side channel question. Consider the following question the analyst has to answer: *Is there a side channel in time in LEEDS that allows an attacker to determine whether the range of values he searches contains a restricted ID?*

Phase I. Our application-agnostic automation generates catalogs for entry points, potential secrets, loops and branches in the application. The analyst observes that there are 8 entry points (7 are main methods and one UDP request handler) and 130 potential secrets (76 fields and 34 callsite returns). The loop catalog reports 106 loops, of which 100 terminate based on Strings or collections, and 17 of the 106 involve events such as network or file I/O. The branch catalog reports 225 branches, of which 75 govern loops or network or file I/O events. The analyst observes that the 17 loops involving network or file I/O could be a good starting point for his Phase II activity of narrowing down program artifacts.

Phase II. In Phase II, the analyst first uses the application's description to identify an entry point and the secret.

Entry Point Identification. LEEDS is a network database server, so the analyst identifies a UDP Request handler as the relevant entry point (other entry points are main methods inaccessible to the clients) for starting his audit.

Secret Identification. To identify the secret, the analyst again leverages knowledge from the application description – that there could be multiple restricted IDs in the LEEDS database, so the secret is likely to be a collection rather than a primitive or a String. Thus, the analyst inspects only the 13 (of the 130) potential secrets that are collections. By manual inspection of the 13 collections, the analyst identifies the field `ids` of type `ArrayList` in the app as the secret (restricted IDs).

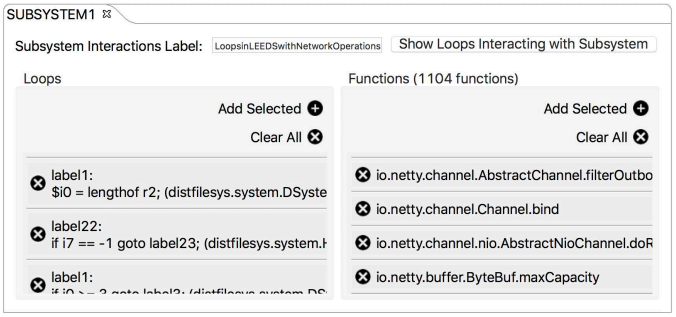

Fig. 6: Subsystems Interactions Filter UI configured with loops and network APIs used in the LEEDS application

Narrow down Program Artifacts. The analyst decides to explore two strategies to narrow down program artifacts: (1) Use LCG to find reachable loops from the `UDPServerHandler.channelRead0` entry point; (2) Since the description says that LEEDS is a network database server, and the question is to find a side channel in time rather than space, use subsystem interactions to select loops (time consuming program artifacts) that govern network operations. The LCG from the entry point of interest (`UDPServerHandler`) selects 99 loops (of the 106 loops in the application), which does not provide sufficient reduction of program artifacts to hypothesize SSCVs. Switching to second strategy, the analyst invokes the Subsystem Interactions Filter and configures it with the 106 loops and 1104 network APIs as input (Figure 6). This results in 14 loops, a significant reduction.

The analyst observes that 8 of the 14 loops, namely $L_1, \ldots L_8$ are present in the entry point method. The analyst further confirms from the loop and branch catalog (generated in Phase I) that the app has a branch condition (b) governing a network write operation as well. At this point, the analyst has narrowed down the program artifacts, and proceeds to scrutinize b, $L_1 \ldots L_8$.

Hypothesizing SSCV Patterns. Observing that b, $L_1 \ldots L_8$ occur in the same method `UDPServerHandler.channelRead0`, the analyst considers plausible SSCV patterns that may be present in the application. Specifically, the analyst has the following questions.

1. Does the branch b create differential paths with and without network operations, indicating the plausibility of SSCV pattern 2?
2. How is branch b related to loop L_i, $i \in \{1 \ldots 8\}$; specifically, do b and any of the L_i's combine to induce differential behavior according to SSCV patterns 1 or 5?

3. Is the branch b predicated on the secret?

Interactive Analysis using PCG. The CFG for `UDPServerHandler`'s `channelRead0` method has 194 nodes and 324 edges, which is difficult to comprehend. To view only the behaviors with respect to network interactions, the analyst decides to inspect the method using a PCG with selected network write operations as events. The analyst uses the following parameters to construct the PCG with respect to L_i : the CFG for the loop L_i's body and the contained network write events. The analyst observes that the resulting PCG with network operation events for one of the loops, say L_1, contains the branch b, so the analyst further scrutinizes the relationship between b, L_1 and network events.

Before proceeding further, the analyst uses an independent taint analysis to confirm that (a) b is predicated on the secret (i.e., there exists a taint from the secret `ids` to the branch condition b), and (b) the loop L_1 iterates for every ID in the database within the search range provided as input. Now the analyst is ready to make a hypothesis.

Figure 7 shows the PCG generated using L_1's loop body and network write event. The top ($\top$) and bottom ($\bot$) nodes in the PCG correspond to the entry and exit for the control flow used to construct the PCG. The cyan node is L_1's loop header, and the yellow node is the selected event (network write operation). The solid and dotted edges from the branch condition nodes (diamonds) correspond to the paths taken when the branch condition evaluates to true and false respectively. For example, loop L_1's header is also its termination branch condition, so when it evaluates to false, L_1 terminates (indicated by dotted edge from loop header to $\bot$).

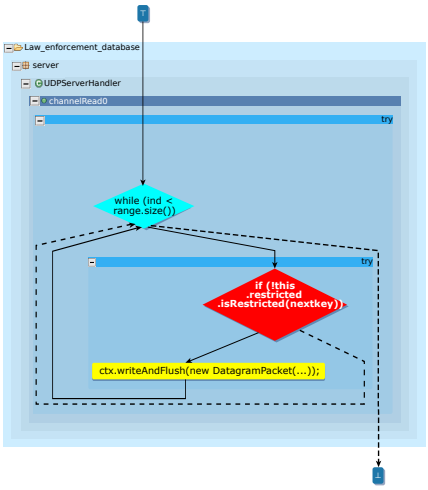

Fig. 7: PCG showing the differential behavior in the LEEDS application

The PCG clearly identifies b (red diamond) as a branch condition governing differential behavior with respect to two paths: the true path contains a network write operation (call to `writeAndFlush`), while the false path does not. Inspecting the code (Listing 1.1), the analyst finds that the other path b (line 6) governs, has another loop L' (lines 13-15). Since b is already within loop L_1, the analyst hypothesizes that b, L_1 and L' combine to induce differential behavior that has traits of three SSCV patterns 1, 2 and 5 (Section 3.2). Specifically, the hypothesis is that L_1 iterates for every ID in the input range, and if it is not a restricted ID (secret), b induces a network event, otherwise b induces a resource consuming operation (loop).

Phase III. Finally, in Phase III the analyst performs dynamic analysis to confirm his hypothesis. The analyst records the timings when searching for a

range of IDs including the restricted ID. The analyst confirms that when the range includes a restricted ID, network packets are received for every ID in the database within the range except the restricted ID. The attacker can deduce that a restricted ID is contained in the input range whenever he observes a longer interval between 2 consecutive network packets received from the server. This confirms the analyst's hypothesis.

Listing 1.1: `UDPServer.channelRead0`: code snippet

```
1  int ind = 0;
2  // Loop L1
3  while (ind < range.size()) {
4   Integer nextkey = range.get(ind); ...
5   // Differential branch condition b
6   if (!this.restricted.isRestricted(nextkey)) { ...
7    // Network write event on the true path from b
8    ctx.writeAndFlush((Object)new DatagramPacket(bos,
          (InetSocketAddress)packet.sender())); ...
9    continue;
10  } ...
11  Integer getkey = range.get(ind);
12  // Time consuming loop L' on false path from b
13  while (this.restricted.isRestricted(getkey) && ind < range.size()) { ...
14   getkey = range.get(ind);
15  }
16 }
```

9 Related Work

Kocher [18] was the first to demonstrate a side channel attack using timing information to expose secret keys used in RSA, Diffie-hellman, Digital Signature Standard, and other cryptosystems. Subsequently, several side channel exploits against cryptographic algorithms on security hardware and cache architectures have been demonstrated [22,21,14,23,15,5,30,16].

In their seminal paper, Brumley and Boneh [8] showed that such side channel attacks also apply to general software systems, and demonstrated a timing side channel attack that could extract private keys from an OpenSSL-based remote web server. Since then, several other side channel attacks have been demonstrated on remote servers and software [26,6,24,20,7,28,25]. Software side channel vulnerabilities (SSCVs) are particularly difficult to detect [3,9]. We discuss related work on the complexity of detecting SSCVs.

Demme et al. introduced Side-Channel Vulnerability Factor (SVF) [12] as a measure of the correlation between execution traces of a microprocessor and an attacker's observation traces. Zhang et al. proposed Cache Side-channel Vulnerability (CSV) [32] as an improvement over SVF specifically to measure cache based side channel vulnerabilities. Köpf et al. [19] model the amount of information about the secret leaked by an application using information-theoretic entropy measures. Doychev et al. [13] perform static analysis to provide a quantitative upper bound on the amount of information contained in various side channel observables such as timing and events corresponding to cache accesses.

The research as noted above pertains to metrics that are specific to certain types of secrets (private keys of cryptosystems), and involve a limited range of

observables (e.g., cache access timing, cache hit or miss events, time spent on certain operations on the CPU). In contrast, as we have described, SSCVs admit much wider variability in terms of the secret and observables. As an example, in our case study (Section 8), the secret was a globally declared collection in the application and the observable was timing of network packets received from the victim.

Sidebuster [31] analyzes Java web applications to detect and quantify potential side channels. Sidebuster requires the developers or analysts to label program artifacts as sensitive, and uses taint analysis to identify branch conditions tainted by them. The taint analysis based detection performed by Sidebuster is subsumed by the analyses supported by our tooling (see Sections 5.2 and 5.3).

As far as we know, the open-ended SSCVs and a systematic human-on-the-loop automation for detecting them is being described for the first time with this paper.

10 Conclusion

Software side channel vulnerabilities (SSCVs) pose a serious threat to cybersecurity. They have challenged modern cryptographic algorithms and have enabled attackers to compromise remote servers and web applications. The possibilities for SSCVs are open-ended, beyond the well-studied cryptographic SSCVs. With that in mind, the US defense research agency DARPA created the STAC program [3] to create an innovative technology to address the threat of SSCV attacks. We as STAC participants involved in developing a such technology are challenged with a mixture of benign and vulnerable software to evaluate how well the human-on-the-loop automation works in practice on large software. This paper presents the research for developing such automation.

Our approach tries to bound the open-ended SSCVs by classifying them with overarching patterns. At the current stage of research we have five overarching patterns. We have created a public repository [4] of representative SSCVs based on these patterns. This repository is meant to serve as valuable examples for other researchers interested in pursuing research on SSCVs. We present application-agnostic automation, one that works across all the overarching patterns, to derive and catalog relevant program artifacts and their attributes. The results of application-agnostic automation feed into the subsequent application-specific interactive automation to hypothesize potential SSCVs and gather evidence to prove their existence with targeted dynamic analysis. We present novel interactive automation using parameterized program graph models.

We present a case study to demonstrate how the proposed human-on-the-loop automation can be used in practice. We bring out its significance by an experimental evaluation using metrics to measure the complexity of detecting SSCVs. An interesting and important area for future research is to scan the open-source software to look for potential SSCVs based on the overarching patterns. It involves difficult challenges of minimizing the human effort while maintaining the accuracy of detecting SSCVs.

Acknowledgements

We thank our colleagues from Iowa State University and EnSoft for their help with this paper. Dr. Kothari is the founder President and a financial stakeholder in EnSoft.

References

1. Klocwork source code analysis. http://www.klocwork.com (2001)
2. Coverity static analysis. http://www.coverity.com (2002)
3. Space/time analysis for cybersecurity. http://www.darpa.mil/program/space-time-analysis-for-cybersecurity (2015), accessed: Mar. 2016
4. Software Side Channel Vulnerabilities Repository. https://github.com/kcsl/SSCV/ (2017), [Online; accessed 18-August-2017]
5. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: Ooh aah... just a little bit: A small amount of side channel can go a long way. In: Cryptographic Hardware and Embedded Systems, pp. 75–92. Springer (2014)
6. Black, J., Urtubia, H.: Side-channel attacks on symmetric encryption schemes: The case for authenticated encryption. In: Proceedings of the 11th USENIX Security Symposium. pp. 327–338 (2002)
7. Bosman, E., Razavi, K., Bos, H., Giuffrida, C.: Dedup est machina: Memory deduplication as an advanced exploitation vector. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 987–1004 (2016)
8. Brumley, D., Boneh, D.: Remote timing attacks are practical. Computer Networks 48(5), 701–716 (2005)
9. Chen, S., Zhang, K., Wang, R., Wang, X.: Side-channel leaks in web applications: A reality today, a challenge tomorrow. 2010 IEEE Symposium on Security and Privacy (SP) 00, 191–206 (2010)
10. Cummings, M.: Supervising automation: humans on the loop. http://web.mit.edu/aeroastro/news/magazine/aeroastro5/cummings.html (2008), [Online; accessed 10-May-2017]
11. Deering, T., Kothari, S., Sauceda, J., Mathews, J.: Atlas: a new way to explore software, build analysis tools. In: Proc. of International Conference on Software Engineering. pp. 588–591. ACM (2014)
12. Demme, J., Martin, R., Waksman, A., Sethumadhavan, S.: Side-channel vulnerability factor: A metric for measuring information leakage. SIGARCH Comput. Archit. News 40(3), 106–117 (Jun 2012)
13. Doychev, G., Köpf, B., Mauborgne, L., Reineke, J.: Cacheaudit: A tool for the static analysis of cache side channels. ACM Transactions on Information Systems Security 18(1), 4:1–4:32 (June 2015)
14. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering pp. 1–27 (2016)
15. Gras, B., Razavi, K., Giuffrida, E.B.H.B.C.: Aslr on the line: Practical cache attacks on the mmu (2017)
16. Gullasch, D., Bangerter, E., Krenn, S.: Cache games–bringing access-based cache attacks on aes to practice. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy. pp. 490–505. IEEE Computer Society (2011)

17. Holland, B., Santhanam, G.R., Awadhutkar, P., Kothari, S.: Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 79–84 (2016)
18. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Advances in Cryptology. pp. 104–113. Springer (1996)
19. Köpf, B., Basin, D.: An information-theoretic model for adaptive side-channel attacks. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 286–296. CCS '07, ACM (2007)
20. Lawson, N.: Side-channel attacks on cryptographic software. IEEE Security & Privacy 7(6) (2009)
21. Matthews, A.: Side-channel attacks on smartcards. Network Security 2006(12), 18–20 (2006)
22. Messerges, T.S., Dabbish, E.A., Sloan, R.H.: Investigations of power analysis attacks on smartcards. In: Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology. pp. 17–17. USENIX Association (1999)
23. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D.: The spy in the sandbox: Practical cache attacks in javascript and their implications. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. pp. 1406–1418. ACM (2015)
24. Polakis, I., Argyros, G., Petsios, T., Sivakorn, S., Keromytis, A.D.: Where's wally?: Precise user discovery attacks in location proximity services. In: Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security. pp. 817–828. ACM (2015)
25. Saura, D., Futoransky, A., Waissbein, A.: Timing attacks for recovering private entries from database engines. Black Hat USA (2007), https://www.blackhat.com/presentations/bh-usa-07/Waissbein_Futoransky_and_Saura/Presentation/bh-usa-07-waissbein_futoransky_and_saura.pdf
26. Song, D.X., Wagner, D., Tian, X.: Timing analysis of keystrokes and timing attacks on ssh. In: Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (2001)
27. Tamrawi, A., Kothari, S.: Projected control graph for accurate and efficient analysis of safety and security vulnerabilities. In: Asia-Pacific Software Engineering Conference (APSEC). pp. 113–120 (Dec 2016)
28. Vila, P., Köpf, B.: Loophole: Timing attacks on shared event loops in chrome. arXiv preprint arXiv:1702.06764 (2017)
29. Wei, T., Mao, J., Zou, W., Chen, Y.: A new algorithm for identifying loops in decompilation. In: Static Analysis, pp. 170–183. Springer (2007)
30. Yarom, Y., Falkner, K.: Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In: Proceedings of the 23rd USENIX Conference on Security Symposium. pp. 719–732. USENIX Association, Berkeley, CA, USA (2014)
31. Zhang, K., Li, Z., Wang, R., Wang, X., Chen, S.: Sidebuster: Automated Detection and Quantification of Side-channel Leaks in Web Application Development. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. pp. 595–606. ACM (2010)
32. Zhang, T., Liu, F., Chen, S., Lee, R.B.: Side channel vulnerability metrics: The promise and the pitfalls. In: Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy. pp. 2:1–2:8. ACM (2013)