TSA: A Tool to Detect and Quantify Network Side-Channels[*]

İsmet Burak Kadron
kadron@ucsb.edu
University of California, Santa Barbara
Santa Barbara, CA, USA

Tevfik Bultan
bultan@ucsb.edu
University of California, Santa Barbara
Santa Barbara, CA, USA

ABSTRACT

Mobile applications, Internet of Things devices and web services are pervasive and they all encrypt the communications between servers and clients to not have information leakages. While the network traffic is encrypted, packet sizes and timings are still visible to an eavesdropper and these properties can leak information and sacrifice user privacy. We present TSA, a black box network side-channel analysis tool which detects and quantifies side-channel information leakages. TSA provides the users with the means to automate trace gathering by providing a framework in which the users can write mutators for the inputs to the system under analysis. TSA can also take as input traces directly for analysis if the user prefers to gather them separately. TSA is open-source and available as a Python package and a command-line tool. TSA demo, tool and benchmarks are available at https://github.com/kadron/tsa-tool.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; **Web application security**; **Network security**; • **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Security and privacy, Side-channel analysis, Network traffic analysis, Software testing

ACM Reference Format:
İsmet Burak Kadron and Tevfik Bultan. 2022. TSA: A Tool to Detect and Quantify Network Side-Channels. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558938

1 INTRODUCTION

Information leaks are becoming a threat on privacy as all sensitive information migrates to online services and side-channel information leakages, where private information can be extracted by analyzing visible side effects of computation, are becoming important. On encrypted network communications, eavesdroppers can utilize communication metadata (packet sizes and timings) to infer the user actions or data on mobile applications, websites or IoT devices [1, 9]. A thorough analysis of these side-channel leakages is needed to check if they leak any important information. White-box analysis of these information leakages are difficult due to non-determinism of the network conditions and difficulty of multi-component static analysis. This work focuses on black-box analysis where profiling the application with many inputs and performing analysis on the resulting network traces is more feasible.

We present TSA, a **T**ool for detecting and quantifying network **S**ide-channels **A**utomatically with black-box testing and input generation methods. TSA is based on the technical approaches presented in [6, 8]. We extend these prior works into a flexible open source tool with documented APIs which the users can utilize to analyze their applications. In network trace analysis, we provide the option of trace alignment to extract more meaningful features for trace analysis. For experimental evaluation and demonstration of the capabilities of TSA, we analyze 3 applications in DARPA STAC benchmark.

TSA can be used in two ways. If the user is testing an application and has not collected any traces, they can use TSA to generate inputs to test the application, capture the network traces, and analyze the captured traces in a feedback driven loop where TSA terminates the analysis if the information leakage estimate converges to a value. If they already captured some traces previously, they can use TSA to just perform side-channel analysis and quantification.

When using TSA, the user provides some seed inputs for the target system, and a set of mutators which, given a valid input, return another one. The user chooses a *secret of interest*—some aspect of the input that they consider sensitive, whose leakage they want to detect and quantify. TSA then repeatedly executes the target system, generates new inputs, captures network traffic, and adjusts input generation strategy based on the feedback it obtains by analyzing captured traffic. For analysis, TSA extracts features that may leak side-channel information using the size, time and direction of the captured network packets. Afterwards, it computes the mutual information using Shannon entropy and finds features that maximize the information gain about the secret of interest. The final output from TSA is an automatically generated *ranking* of the top n most-leaking features, sorted by how much information they each leak about the secret of interest.

The *envisioned users* of TSA include researchers and software engineers, and other people who want to analyze the side-channel information leakage of their applications. The *challenge* we propose to address is automatically analyzing side-channel information leakages of applications using a small set of inputs and mutators. In Section 2, we summarize the differences between our approach and the prior work. In Section 3, we go over the tool architecture and API to describe how it can be used for analysis. In Section 4,

[*]This material is based on research supported by NSF under Grants CCF-1901098 and CCF-1817242.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3558938

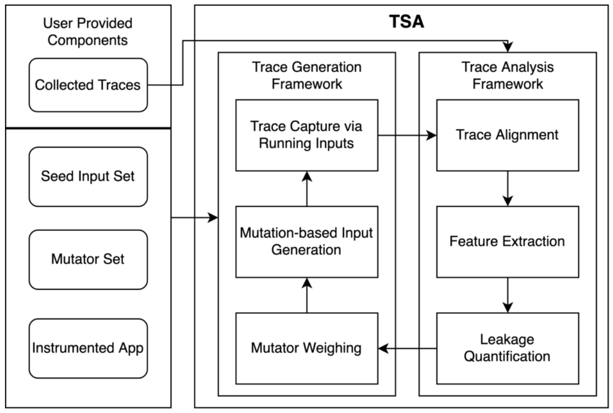

Figure 1: Architecture of TSA.

we describe the results of our *experimental evaluation*. In Section 5, we conclude the paper.

2 RELATED WORK

There have been prior works for analyzing side-channel information leakages on encrypted network traces such as LeakiEst from Chothia et al. [3], F-BLEAU from Cherubin et al. [2] and WeFDE from Li et al. [7]. LeakiEst uses histograms to estimate the probability distributions and uses min-entropy measure to find the worst case information leakages on single features. F-BLEAU uses nearest neighbor classifiers to quantify the information leakage over all features. WeFDE uses KDE with Monte Carlo sampling to quantify the information leakage over all features. All of these approaches provide information leakage quantification capabilities given extracted features and labeled traces with various probability estimation and different quantification measures. In addition to quantifying the information leakage with a dynamic probability estimation method similar to WeFDE, TSA uses user provided inputs and mutators to automate the input generation and trace capture in a feedback-driven manner.

3 TSA

In this section, we describe TSA's architecture and main workflow of TSA's execution. We also describe how TSA is used on an example application, how the user provides inputs, mutators and application orchestration.

3.1 TSA Framework

Figure 1 describes the core framework of TSA and two ways the user can provide data for the analysis. If the user provides seed inputs, mutators and an instrumented application, TSA uses its *trace generation framework* to generate new inputs and run those inputs to generate network traces. TSA's *trace analysis framework* takes the generated traces and performs side-channel analysis with trace alignment, feature extraction and quantifies the information leakage over each feature. The leakage quantification results are used to determine the importance of mutators which are used to generate new inputs in mutation-based input generation. If the user only provides collected and labeled network traces, TSA's *trace analysis framework* performs side-channel analysis as described and returns the leakage quantification results.

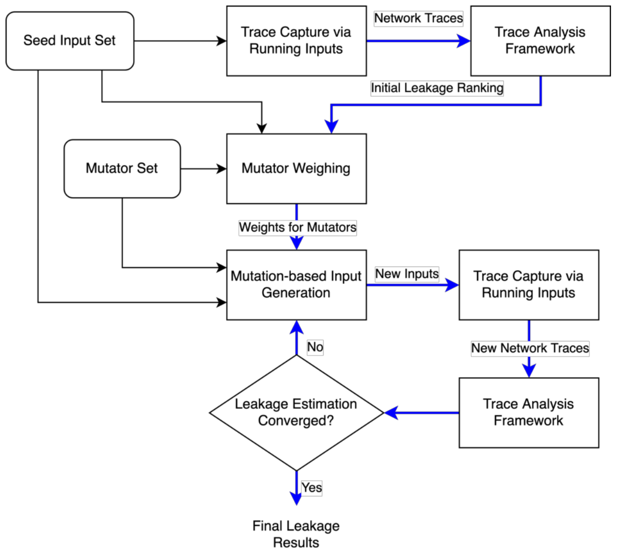

Figure 2: Workflow of TSA**. Thick blue arrows denote the flow of execution.**

TSA Workflow Summary. Figure 2 describes the workflow of TSA when used with a set of seed inputs and mutators. We only describe the workflow of this use as TSA's workflow with a set of collected network traces is explained clearly in Figure 1 and in the previous section. To obtain an initial leakage estimation, TSA runs the seed inputs over the instrumented application to obtain an initial set of traces, uses trace alignment and feature extraction to obtain features and use quantification methods to quantify the information leakage. Using this initial leakage estimation, TSA evaluates the influence of mutators on the leakage estimation based on changes in top feature or secret and computes weights for mutators which are proportional to their likelihood of changing secret value or perturbing feature values.

After this initial setup, for each iteration, TSA generates new inputs using mutators and previous inputs based on the computed weights, runs these new inputs over the system to obtain new traces and runs the analysis over all of the collected traces to obtain the leakage estimation for that iteration. If the stopping criterion is satisfied, then it returns the final information leakage estimation on the application. Otherwise, it starts a new iteration, repeating the previously described steps. We perform analysis over all the collected traces and generate new inputs using all the previously generated inputs but we do not show the accumulation of traces and inputs on the figure for simplification.

3.2 TSA API

To analyze their applications, users need to define the input model, provide a set of mutators and write code to orchestrate system setup and execution. To make this process easier, we provide an

API with classes for defining inputs, mutators and application orchestration. Listing 1 provides an example code segment the user may write to test an example shopping application extending TSA API classes. The TSA codebase contains examples with varying degrees of complexity, including apps, inputs, and mutators for the STAC benchmark which the users can refer to as well.

Input model. To help the users write inputs, we provide `Input` base class which represents a valid input for the application. Users can subclass `Input` and add fields and members to model the relevant characteristics. For example, if the user wants to test if their purchases are leaked, user can define a shopping list input as Python class `ShoppingInput` extending `Input` with a list named `shoppingcart` representing their purchases and string named `zipcode` representing the ZIP code of their shipping address. The users can also write assertions in the constructor such as ZIP code belonging to a set of valid US ZIP codes to check validity of the input when it is being created.

Listing 1: Example usage of TSA **API**

```
from tool import Platform, Container, Sniffer, App, Input, Mutator

class ExampleApp(App):
    def launch(self):
        Platform.cleanuphosts(["homer.example.edu", "marge.example.edu"])
        # Deploy two containers on two different machines
        self.servercontainer = Container("example/server:v1.0")
        self.clientcontainer = Container("example/client:v1.0")
        self.server = Platform.launch(self.servercontainer, "homer.example.edu")
        self.client = Platform.launch(self.clientcontainer, "marge.example.edu")
        # Run the server
        server_cmd = "bash -c 'cd /home/server && ./startServer.sh'"
        self.server.exec(server_cmd, detach=True)
    def shutdown(self):
        self.server.killrm()
        self.client.killrm()
    def run(self, inputs):
        sniffer = Sniffer(ports=[8080, 8081])
        sniffer.start()
        for input in inputs:
            sniffer.startinteraction(input.secret())
            self.client.createfile(input, "/home/client/input.txt")
            cmdfmt = "bash -c 'cd /home/client && ./startClient.sh {} {}'"
            self.client.exec(cmdfmt.format("homer.example.edu", "input.txt"))
        sniffer.stop()
        return sniffer.traces()

class ShoppingInput(Input):
    def __init__(self, shoppingCart, zipcode):
        assert len(shoppingCart) > 0
        assert len(zipcode) == 5
        self.shoppingCart = shoppingCart
        self.zipcode = zipcode
        self.itemList = ['apple', 'orange', ...]
    def __eq__(self, other):
        return self.shoppingCart == other.shoppingCart
            and self.zipcode == other.zipcode
    def __hash__(self):
        return hash((self.shoppingCart, self.zipcode))
    def secret(self):
        return len(self.shoppingCart)

class AddItem(Mutator):
    def mutate(input):
        randomItem = random.choice(input.itemList)
        input.shoppingCart.append(randomItem)
        return input

class RemoveItem(Mutator):
    def mutate(input):
        if len(input.shoppingCart) > 0:
            randomIndex = random.randrange(len(input.ShoppingCart))
            input.shoppingCart.pop(randomIndex)
            return input
        else:
            return None

class ChangeZIPCode(Mutator):
    # ... etc ...
```

The only mandatory methods to implement are methods `hash` and `secret`. First method `hash` is used by TSA to check if the newly generated inputs are unique. A simple way to implement `hash` is to pack all relevant class members in a tuple and call Python's primitive `hash` method on that tuple. This ensures that any change in any member affects the resulting hash value. The second method `secret` defines the secret of interest in relation to the input. This is up to the user and in our example, it can be number of elements in the shopping cart, the total cost of all items in the shopping cart, prefix of user's ZIP code (denoting general area of delivery), or any other sensitive information.

Mutators. `Mutator` base class in TSA API represents a mutator that transforms valid inputs. The users can write their own mutators extending `Mutator` and providing their own implementation for the method `mutate`, which is a static method that takes an `Input` and returns another `Input`. The method assumes `Input` is a valid input for the system and tries to return another valid input. If it cannot, the method should return `None`. For example, the user may write a mutator which adds an item to the shopping cart or another mutator which removes an item from the shopping cart if possible.

Listing 1 shows an example with three mutators. The first mutator, `AddItem`, adds an item to the shopping cart field and returns the new input. The second mutator, `RemoveItem`, removes a random item from the shopping cart field if it is not empty, and returns the new input. If the shopping cart is empty, it returns `None` as it cannot remove items from an empty list. The third mutator, `ChangeZIPCode`, changes the ZIP code of the input from a set of valid ZIP codes, returning the new input. We provide the code for the first two mutators for space reasons as the code to check valid ZIP codes is complicated.

System Setup and Execution. To execute the system under test, we provide `App` base class which can be extended by the users. When implementing the instrumentation of system execution, the user must implement three methods, `launch`, `shutdown` and `run`. `launch` and `shutdown` methods set up the system before analysis and shut down the system after analysis respectively. `run` method takes a list of `Input` objects and runs them one by one over the system under test, returning a set of captured network traces. The user needs to provide how the inputs interact with the system by implementing `run` method. This imitates how a user might use the input as a scenario. For example, for `ExampleApp.run()` might have a script that searches each item of the input file on a website, puts the first result on the shopping cart and checks out using the ZIP code in the input.

To instrument deployment and launching of components such as clients, servers and peers, we use Docker [5] in our examples. We provide a classes called `Container` and `Platform` which set up and launch Docker containers respectively. We provide methods to create containers on hosts and on the containers, we provide methods to copy files, run commands and shutdown. Using Docker is optional, but recommended for simplicity and reproducibility. This also allows running TSA analyses on cloud platforms with minimal changes.

Packet sniffing. To help with capturing traffic, TSA provides a `Sniffer` class that offers a simple interface for capturing traffic and labeling the captured traces. To set up network capture, the user can create a Sniffer object, denoting specific ports they want to listen and start the sniffing which runs in a separate thread. Before starting each interaction, the App's `run` method should call `Sniffer.startinteraction(secret)` to ensure that the captured traffic is labeled with the correct secret. Lastly, `run` should finish sniffing with `Sniffer.stop()` method and return the traces obtained from `Sniffer`.

TSA *Setup.* Our tool runs in a feedback-driven manner, where it generates inputs by picking mutators based on a heuristic, generates new traces by running the inputs on the instrumented app and runs our analysis on the newly obtained traces. If the leakage estimation of top-k features do not change below an ϵ value for N steps, then the estimation stops. To setup this feedback loop, we provide default values to the stop criterion parameters but the users can provide their own values for k, ϵ and N variables to set up their own stop criterion. Users can also provide a parameter to determine how many times each input will run on the system. Some systems may exhibit non-deterministic behaviors, therefore running each input multiple times may be beneficial for the accuracy of the analysis.

3.3 TSA Usage

TSA[1] is available as a command-line tool and Python package. As a Python package, TSA can be used as a library that provides classes for sniffing network communication, parsing network traces and extracting features, quantifying information leakage and visualizing the feature distributions and information leakage. Defining input models, mutators, system instrumentation and providing seed inputs require writing them in Python, therefore this is the *recommended* way to use TSA for *feedback-driven analysis.* We provide examples on how the TSA is used as a Python package in our repository.

TSA's command-line interface is used for analyzing already captured network traces. TSA's command-line arguments include network trace and label file names, which ports to examine for filtering traffic, and folder location for generated plots. There are flags for choosing the leakage quantification options, choosing whether to use alignment on network traces, and whether to quantify only space or time features if the user is interested in only one of them.

In both usages, TSA outputs the leakage information as a ranking over the extracted features. If requested, TSA also provides plots for the feature distributions per secret to show how features and secrets correlate.

4 EVALUATION

To demonstate the performance of our approach when input sets and mutators are given, we used TSA to analyze information leakages of 2 applications in the DARPA STAC benchmark [4] which contain implementations of various client-server or peer-to-peer web applications such as a messaging app, or a railyard or air traffic management system. The applications we analyzed are Airplan, Railyard and GabFeed. This benchmark comes with some ground truth where some versions are found to be leaking information through manual analysis which we can compare our results against. We report the leakages in terms of percentages and amount of bits leaked compared with the full amount of information of the secret set. Overall, the iterative analysis took at most 5 hours for all cases.

Airplan Results. For Airplan, an air traffic management system which takes a route map graph as an input, with number of airports as the secret of interest, we provided 13 seed inputs corresponding to one for each secret value (a graph with 2 nodes, 3 nodes, etc.) and 12 mutators which add/remove nodes, add/remove flights, modify airport names and each weight separately.

Using our tool, we find that Airplan 2, the vulnerable application leaks 100% (3.70 out of 3.70 bits) of the information within 76 minutes of analysis and 3 iterations. Airplan 5 is a modified version of Airplan 2 where the vulnerability is patched and TSA reports that it leaks 89% (3.29/3.70 bits) of the information within 114 minutes of analysis. Airplan 3 is marked not vulnerable in the DARPA STAC benchmark and TSA reports that it leaks 47% (1.74/3.70 bits) of the information within 161 minutes of analysis. These results are consistent with the ground truth where the vulnerable application leaks 100% and other versions have less leakage depending on different versions.

Railyard Results. For Railyard, a train station management system, we provided 64 inputs with different configurations denoting possible combinations of possible cargo (which is the secret of interest) and 11 mutators that add/remove a train car, a piece of cargo, a crew member, or a stop, change names of crew or stops. Our analysis shows that this application leaks 22% (1.32/6.00 bits) of the information within 202 minutes which is similar to the coarse ground truth provided by DARPA STAC benchmark where it is marked non-vulnerable. There is no vulnerable version of Railyard in the DARPA STAC benchmark to compare against but 22% leakage shows that the application does not leak a significant amount of information.

GabFeed Results. For GabFeed, a social messaging tool where the secret of interest is the Hamming weight or number of 1's in the binary representation of the secret key used in login, we provided 5 mutators modifying number of 1's in the secret key and shuffling the binary representation to generate a new input. We provided 16 random inputs as seed inputs as well. For the leaking version of the application, GabFeed 1, we find that it leaks in average 98% (5.50/5.61 bits) of information within 108 minutes of analysis which is consistent with the ground truth. For the non-leaking versions, GabFeed 2 and GabFeed 5, TSA runs longer, exploring more secrets and finds that they leak 31% (1.86/6.00 bits) of the information with 297 and 240 minutes of analysis time respectively, demonstrating the low amount of leakage.

5 CONCLUSION

We presented TSA for automatically detecting and quantifying network side-channel information leakages. In our presentation, we described how TSA works given inputs and mutators, and how its API can be modified to analyze other applications. In our experimental evaluation, we showed TSA's performance against an existing benchmark.

[1]The tool's source code, experimental evaluation code, evaluation results, and documentation are publicly available at https://github.com/kadron/tsa-tool

REFERENCES

[1] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. 2020. Peek-a-Boo: I see your smart home activities, even encrypted!. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 207–218.

[2] Giovanni Cherubin, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. 2019. F-BLEAU: Fast Black-box Leakage Estimation. *CoRR* abs/1902.01350 (2019). arXiv:1902.01350 http://arxiv.org/abs/1902.01350

[3] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. 2013. A Tool for Estimating Information Leakage. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 690–695. https://doi.org/10.1007/978-3-642-39799-8_47

[4] DARPA. 2015. *The Space-Time Analysis for Cybersecurity (STAC) program.* http://www.darpa.mil/program/space-time-analysis-for-cybersecurity

[5] Docker Inc. 2013. *Docker SDK and API.* Retrieved June 14, 2022 from https://docs.docker.com/engine/api/sdk/

[6] Ismet Burak Kadron, Nicolás Rosner, and Tevfik Bultan. 2020. Feedback-Driven Side-Channel Analysis for Networked Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis.*

[7] Shuai Li, Huajun Guo, and Nicholas Hopper. 2018. Measuring Information Leakage in Website Fingerprinting Attacks and Defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1977–1992. https://doi.org/10.1145/3243734.3243832

[8] Nicolás Rosner, Ismet Burak Kadron, Lucas Bang, and Tevfik Bultan. 2019. Profit: Detecting and Quantifying Side Channels in Networked Applications. In *26th Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019.*

[9] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. 2018. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security* 13, 1 (Jan 2018), 63–78. https://doi.org/10.1109/TIFS.2017.2737970