ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State

Jeongseok Son
Griffin Prechter
Rishabh Poddar
Raluca Ada Popa
Koushik Sen

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-29
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-29.html

May 1, 2021

ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State

by Jeongseok Son

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Raluca Ada Popa
Research Advisor

(Date)

* * * * * * *

Professor Koushik Sen
Second Reader

(Date)

Acknowledgement

I still remember the day when Raluca reached out to me for an admission interview. I already knew her famous CryptDB, one of the most fascinating security works I have ever read. I was excited about exploring a new area with her and fortunately got an acceptance letter from Cal. However, it was not until I met Raluca in person that I made the decision. During the visit day, the RISELab admits and members took a short hike to a faculty's house on a hill. While everyone was chatting and enjoying the panoramic scenery, Raluca quietly filled the cups on a table with water for others. Even within the short visit days, I was able to recognize that she is not just an extremely smart scholar, but also a kind and caring person. After spending three years with her, I cannot have more confidence to say the observation was correct. She guided me to become a better professional by always encouraging me to aim for the highest standard. At the same time, she gave me her honest and careful advice on any matters I brought up to her and was always supportive of whatever decision I made. The best thing I earned throughout my graduate study is not the expertise in computer security. It is the attitude towards my work and other people that I learned from my advisor. Thank you, Raluca.

Koushik and I first met at a RISE retreat. I was struggling to discover a new research idea at that time. Koushik was joining RISELab as a faculty member around then. I was interested in programming languages so I purposefully approached him. I was nervous about talking to a world-renowned expert without knowing much about his area. Surprisingly, Koushik was kind enough to show interest in collaborating with me. That was the moment when our ObliCheck project took off. Even though I was not his advisee technically, Koushik was always there when I needed him. Without his kindness and his practical projects that we built upon, our project would have not been in publishable shape. When I hold a more senior role in my career, I want to be approachable and humble like him.

I also want to thank Ion Stoica for giving me a chance to work with him when I was a first-year. As a new graduate student, it is not always easy to approach an eminent professor like Ion. He carved his time out of his hectic schedule and regularly come to his students first and made himself available and responsive. I will ruminate on his insightful advice and thoughts he shared with me even after graduating.

I will miss my student colleagues most after graduating. It will be impossible to find a place like UC Berkeley, especially RISELab, where I was surrounded by the brightest and energetic friends I have ever interacted with. First of all, I am grateful to Griffin Prechter and Rishabh Poddar for working with me on ObliCheck early on. So many parts of the project were uncertain when we started it. Their contributions were the most crucial catalyst for our work. I appreciate Chia-Che Tsai for being a both good mentor and friend when I worked with him on Civet. It was a great pleasure to have a group lunch every week and discuss various topics in security with my sister colleagues including Weikeng Chen, Ankur Dave, Yuncong Hu, Sam Kumar, Pratyush Mishra, Rishabh Poddar, Jean-Luc Watson, Wenting Zheng, and others. I was lucky to have a set of convivial friends in the RISELab including Rolando Garcia, Jack Kolb, Eric Liang, Richard Liaw, Romain Lopez, Stephanie Wang, Michael Whittaker, and Zongheng Yang. I enjoyed working with Eric Love and Frederik Ebert to host Bar Nights and a Football Night as a member of CSGSA in my first year. My Korean EECS friends, Edward Kim and Dayeol Lee, and I came to Berkeley at the same time. We quickly became close friends sympathizing with each other in the same boat. I wish all my colleagues at UC Berkeley the very best luck and hope our future paths will cross again. Thank you all.

ObliCheck: Efficient Verification of Oblivious Algorithms with Unobservable State

Jeongseok Son Griffin Prechter Rishabh Poddar Raluca Ada Popa Koushik Sen
University of California, Berkeley

Abstract

Encryption of secret data prevents an adversary from learning sensitive information by observing the transferred data. Even though the data itself is encrypted, however, an attacker can watch which locations of the memory, disk, and network are accessed and infer a significant amount of secret information.

To defend attacks based on this access pattern leakage, a number of oblivious algorithms have been devised. These algorithms transform the access pattern in a way that the access sequences are independent of the secret input data. Since oblivious algorithms tend to be slow, a go-to optimization for algorithm designers is to leverage *space unobservable to the attacker*. However, one can easily miss a subtle detail and violate the oblivious property in the process of doing so.

In this paper, we propose ObliCheck, a checker verifying whether a given algorithm is indeed oblivious. In contrast to existing checkers, ObliCheck distinguishes observable and unobservable state of an algorithm. It employs symbolic execution to check whether all execution paths exhibit the same observable behavior. To achieve accuracy and efficiency, ObliCheck introduces two key techniques: Optimistic State Merging to quickly check if the algorithm is oblivious, and Iterative State Unmerging to iteratively refine its judgment if the algorithm is reported as not oblivious. ObliCheck achieves $\times 4850$ of performance improvement over conventional symbolic execution without sacrificing accuracy.

1 Introduction

Security and privacy have become crucial requirements in the modern computing era. In order to preserve the secrecy of sensitive data, data encryption is now widely adopted and prevents an adversary from learning secret information by observing the data content. However, attackers can still infer secret information by observing access patterns to the data. Even though the data itself is encrypted, an attacker can watch which locations of the memory, disk, and network are accessed. Such concerns are growing with the increasing adoption of hardware enclaves such as Intel SGX [57], which provides memory encryption but does not hide accesses to memory. By simply observing the access patterns, many works [7,28,46,50,54,64,65,83] have shown that an attacker can reconstruct secret information such as confidential search keywords, entire sensitive documents, or secret images.

As a result, a rich line of work designs *oblivious* execution to prevent such side channels based on access patterns. There are two types of oblivious algorithms. The first, Oblivious RAM (ORAM) [37,77], can be used *generically* to hide accesses to memory, and fits best workloads of the type "point queries".

Intuitively, ORAM randomizes accesses to memory. However, even the fastest ORAM scheme incurs polylogarithmic overhead proportional to the memory size per access, which becomes prohibitively slow for processing a large amount of data as in data analytics and machine learning. For these workloads, instead, researchers have proposed a large array of *specialized oblivious* algorithms, such as algorithms for joins, filters, aggregates [6,11,15,24,64,79,89], and machine learning algorithms [43,44,55,65,66,74]. These specialized algorithms work by accessing memory according to a *predefined schedule of accesses*, which depends only on an upper bound on the data size and not on data content.

Oblivious algorithms of both types tend to be notoriously slow (e.g., hundreds of times for data analytics [89] and tens of times for point queries [77]). To reduce such overhead, many oblivious algorithms take advantage of an effective design strategy: they leverage *special regions of memory* that are *not observable* to the attacker. Such unobservable memory, albeit often smaller than the observable one, allows the algorithm to make direct and fast accesses to data. It essentially works as a cache for the slower observable memory, which is accessed obliviously. Different works choose different resources as unobservable. For example, some works [6,59,65,69] treat registers as unobservable but all the cache and main memory as observable in the context of hardware enclaves such as Intel SGX. GhostRider [53] employs an on-chip scratchpad as an unobservable space to make the memory trace oblivious. Certain works focus on the network as being observable by an attacker and the internal secure region of a machine as unobservable [64,89]. These works report one or more orders of magnitude [89] performance improvement by leveraging the unobservable memory.

While generic algorithms like ORAM can be heavily scrutinized, specialized algorithms designed for all sorts of settings do not receive the same level of scrutiny. Further, these algorithms can be quite complex, balancing rich computations with efficiency. The designer can miss a subtle detail and violate the oblivious property. Currently, an oblivious algorithm comes with written proof, and users must verify the proof manually. As a result, recent works devise ways to check whether an algorithm is oblivious in an automated way (by looking for a secret dependent branch) using taint analysis [16,39,68,88]. These techniques, however, cannot capture unobservable state and would classify a algorithm as not oblivious because of its non-oblivious accesses to unobservable state. Thus, they cannot model a vast array of modern oblivious algorithms.

We propose *ObliCheck*, a checker that can verify oblivious algorithms having unobservable state in an efficient and accurate manner. ObliCheck allows algorithm designers to write an oblivious algorithm using the APIs to distinguish between observable and unobservable space. Based on this distinction, ObliCheck precisely records the access patterns visible to an attacker. Then, ObliCheck automatically proves that the algorithm satisfies the obliviousness condition. Otherwise, ObliCheck provides counterexamples – i.e., inputs that violate the oblivious property – and identifies program statements that trigger non-oblivious behavior.

ObliCheck primarily aims to verify the oblivious property of the algorithms, not the actual implementations of oblivious programs. ObliCheck employs a Satisfiability Modulo Theories (SMT) solver for symbolic execution and verificaiton. SMT solvers can only solve formulas within a first-order logic theory. Hence, ObliCheck cannot check an arbitrary program. Instead, ObliCheck supports a restricted subset of Javascript as a modeling language. The choice of Javascript is for leveraging an existing program analysis framework, Jalangi [71], for its implementation. We expect an algorithm designer describes the algorithm using ObliCheck APIs to check mistakes and bugs introduced in the algorithm design stage.

1.1 Techniques and contributions

Our first observation is that taint analysis used in prior work [16, 39, 68, 88] is too ‘coarse’ to capture unobservable state. With taint analysis, if a branch predicate contains tainted variables, then a checker simply rejects the algorithm even if both execution paths of the branch display the same observable behavior. Instead, we observe that we can overcome the limitations of taint analysis with symbolic execution [18, 47]. Using symbolic execution, ObliCheck can analyze an input algorithm with unobservable state in a finer-grained manner and reason about how observable and unobservable state changes in each execution path. Even if a branch depends on a secret input variable, ObliCheck correctly classifies an algorithm as oblivious if the two execution paths after the branch show the same observable behavior. For example, if the two paths both send an identically-sized encrypted message over the network, our checker can conclude both branches maintain the same observable state (the size of the message and its destination) since the message content itself is encrypted (thus unobservable).

However, a naïve application of symbolic execution does not scale. The main challenge with employing symbolic execution is that the program state quickly blows up as the number of branches in the program increases, making it infeasible to complete the check for many algorithms. While traditional state merging [10, 32, 33, 36, 73] can merge states to alleviate the path explosion problem to some extent, it only works when the values in two different paths are the same. To address this problem, ObliCheck employs its *optimistic state merging* technique (§4), which leverages the domain-specific knowledge of oblivious algorithms that the actual values are unobservable to the attacker. ObliCheck uses this insight to optimistically merge two different values with different path conditions by introducing a *new* unconstrained symbolic value for over-approximating the original symbolic variable.

Such “aggressive” state merging for symbolic values is effective at tackling path explosion, but can result in a false “not-oblivious” prognosis. If a symbolic variable, x, is merged into an unconstrained new symbolic variable y, later accesses to y in a conditional statement may trigger an execution path which would have been impossible if x were not replaced with unconstrained y. To address this issue, we devise a technique called *iterative state unmerging* (§5). ObliCheck records symbolic variables merged during the execution. Then, it iteratively refines its judgment by backtracking the execution and unmerges a part of merged variables which may have caused the wrong prognosis. This iterative probing process continues until it either classifies the algorithm as oblivious, or completes the refinement process.

Although optimistic state merging followed by iterative state unmerging costs extra symbolic execution, we found that the overhead is tolerable. This is because our target algorithms are *mostly* oblivious: an algorithm designer who wants to check their algorithm for obliviousness likely did a decent job making much of the algorithm oblivious, but is worried about subtle mistakes. Hence, most algorithms do not require the iterative state unmerging process, and even when an algorithm needs the extra runs, our evaluation shows that the overhead is less than 70% of single execution time. Further, when ObliCheck reports an algorithm as not oblivious, ObliCheck produces the counterexamples that violate the obliviousness verification condition. This information provides valuable help to the algorithm designers to amend their algorithm.

Finally, a well-known limitation of symbolic execution is its inability to verify an algorithm containing an input-dependent loop, requiring the user to provide loop invariants manually, making it hard to verify oblivious algorithms written in terms of an arbitrary length of the input. In ObliCheck, we design a loop summarization technique (§6) that can automatically generate a loop invariant for common loop patterns employed in oblivious algorithms: each iteration of a loop appends a constant number of elements to the output buffer. Using this observation, ObliCheck can automatically figure out the side-effect of a loop on the output length, enabling it to verify oblivious algorithms not tied to a concrete length of the input.

We evaluated ObliCheck using existing oblivious algorithms, and find that ObliCheck improves the verification performance up to $\times 4850$ over conventional techniques. The checking time of ObliCheck grows linearly as the number of input records grows, whereas that of an existing technique increases exponentially.

2 Background and Existing Approach

We first provide necessary background information regarding the oblivious property and symbolic execution to understand the problems. We then point out the limitations of an existing approach to motivate our approach.

2.1 Oblivious Property and Oblivious Algorithms

The oblivious property implies the access sequences of an algorithm are independent from the secret input data. To achieve the oblivious property in a practical sense, specialized oblivious algorithms have recently been devised. In contrast to Oblivious RAM (ORAM), which compiles a general algorithm and run it in an oblivious manner, oblivious algorithms are designed for a specific purpose for data processing such as distributed data analytics [64, 89], data structures [27, 38, 82], and machine learning [63, 65]. Instead of randomly shuffling and re-encrypting data as ORAM does, oblivious algorithms implement fixed scheduling independent of secret input data in a deterministic manner.

Oblivious algorithms leverage *unobservable space*, a secure region of registers or memory which an attacker cannot observe. Since the unobservable space is not visible to an attacker, an algorithm can access data inside the unobservable space fast in a non-oblivious way. Existing oblivious algorithms use different types of unobservable space to protect secret data from different types of attackers. For example, oblivious algorithms for distributed data analysis [15, 64, 89] assume a network attacker who can observe network traffic but cannot observe a part of local memory. The network attacker can only watch encrypted messages sent over network so the information the attacker can utilize is the network access patterns including the size of the messages and the source and destination network addresses. On the other hand, other works focusing on local data processing [6, 59, 65] regard registers as unobservable space and treat cache and local memory as observable by a memory attacker. We will discuss how ObliCheck captures different threat models under an observable and unobservable space abstraction in §3.1.

2.2 Symbolic Execution and Path Explosion Problem

Symbolic execution runs a program with *symbolic values* as input where symbols represent arbitrary value. Symbolic input is used to analyze the conditions on input values that exercise each part of a program. Throughout the execution, values derived from the input symbols become symbolic expressions containing input symbols. When a conditional statement regarding these symbolic values is encountered, both the `then` and `else` branches are explored unlike normal execution. Now each path has different constraints over the input symbols. This constraint is called *path condition*, and symbolic execution keeps the track of path conditions as it encounters conditional statements during the execution. At the end of the execution, a constraint solver solves the path condition of each execution to generate a set of representative inputs that exercise every path of a program. Symbolic execution has been widely adopted to create complete or high coverage test input sets and researchers have developed symbolic execution frameworks such as Jalangi [71] and KLEE [17].

One of the most common problems that a user of symbolic execution encounters is path explosion. A traditional symbolic execution diverges into two runs for every branch in the code. Thus, the number of paths explored and the corresponding state of symbolic values grows exponentially in the number of branches. In order to alleviate this issue, numerous state merging techniques [10, 32, 33, 36, 73] have been devised. State merging techniques merge the symbolic values changed by the branch statements at join points after each branch. The two diverged paths are converged into one path in this way and thus reduces the number of operations and state maintained after a branch statement. However, this comes at the cost of more complicated path conditions, which increase the solver time spent in a constraint solver.

Check Result	$Algorithm_0$ is actually:	
	Oblivious	Not Oblivious
Accept $Algorithm_0$	True Negative(✓)	False Negative(✗)
Reject $Algorithm_0$	False Positive(✗)	True Positive(✓)

Table 1: Definition of the correct and erroneous classification types of an oblivious checker. Rejecting a benign oblivious algorithm is a false positive case (Type I error). Accepting a not oblivious algorithm is a false negative case (Type II error).

2.3 Existing Approach Using Taint Analysis

Several techniques have been devised to check the access pattern leakage of an algorithm. The most widely used technique is taint analysis. This line of work identifies variables whose values depend on secret input. They track the taints of variables propagated from secret inputs. In this way, a checker can check whether a given algorithm includes a secret dependent branch [16, 68]. Algorithms with secret dependent branches are rejected in this approach assuming that those branches incur information leakage because of the different behaviors in the true and false blocks of the conditional statements.

Limitation. However, if an algorithm designer assumes the network attacker discussed in §2.1 as a threat model, taint analysis can classify a benign oblivious algorithm as not oblivious. The network attacker can only observe the network access patterns including the size of data sent over the network, but not the actual content of the data (which is encrypted). As we defined in Table 1, this is a false-positive error.

Listing 1 shows one example algorithm where the assumption in the existing approach results in a false positive. In this example, `secretInput` is secret input. The predicate (Line 4) contains a secret variable `secretInput[i]`. Hence, taint tracking based techniques reject this algorithm due to the secret branch. However, since the threat model in oblivious algorithms assume the actual content (`(secretInput[i], 0)` in Line 5, `(secretInput[i], 1)` in Line 7) of data is en-

Name	Arguments	Description	Effect
observableWrite	(space, addr, buf)	Write buf at the addr of observable space	τ_P += (<space.ID,W>, addr, size(buf)), space.store[addr] = *buf
observableRead	(space, addr, buf)	Read size(buf) of bytes at addr of observable space	τ_P += (<space.ID,R>, addr, size(buf)), *buf = space.store[addr]
readSecretInput	()	Introduce a secret input	A new tainted symbolic value is added
readPublicInput	()	Introduce a public input	A new untainted symbolic value is added

Table 2: API of ObliCheck. *write*, *read*, *send*, and *recv* are used to describe communication between observable and unobservable space. The first field of a triplet added to the access sequence contains the enumerated type of access of MW, MR, NS, and NR, which encode memory write, memory read, network send and network receive respectively. *readSecretInput*, and *readPublicInput* are necessary to make ObliCheck distinguish the secret inputs from public inputs (Refer to Figure 3).

```
function tag(secretInput, threshold) {
    var buf = [];
        for (var i = 0; i < secretInuput.length;
             i++) {
            if (secretInuput[i] < threshold) {
                buf.push(Pair(secretInuput[i], 0));
            } else {
                buf.push(Pair(secretInuput[i], 1));
            }
        }
    }
    var encrypted = Crypto.encrypt(buf);
    socket.send(ADDR, encrypted);
}
```

Listing 1: An example code from Opaque [3] in Javascript. It tags each element in the secret input and sends the encrypted result over the network. Red varialbes are tainted variables from the secret input secretInput[i]. Since the algorithm has a secret (secretInput[i]) dependent branch, taint analysis based techniques deem that this code has leakage although the observed size of the data (encrypted) does not depend on the secret input.

crypted, both branch blocks have indistinguishable behavior to an attacker. Hence, the example algorithm is oblivious.

Requirements. A more accurate checker for oblivious algorithms should satisfy the following requirements.

1) Be aware of which state of a program is observable or not to an attacker (e.g., in Listing 1, the data content is encrypted, thus invisible, but the size of the data is revealed).
2) Understand the behavior of a program on different execution paths across the whole input space to make a sound judgment of whether an algorithm is oblivious.
3) Know which input values are secret or public to decide the behavior of a program is independent of secret input.
4) Since a checker has a limited time budget, the checking process should be scalable in terms of the number of input data records.

3 ObliCheck Overview

In order to check oblivious algorithms with unobservable state and overcome the limitations of existing approaches, we propose ObliCheck. We now provide an overview of ObliCheck's API, the threat model it assumes, and its security guarantees.

Function	Implementation using ObliCheck API
send(dst, buf)	observableWrite(network, <host, dst>, buf)
recv(src, buf)	observableRead(network, <src, host>, buf)
write(dst, buf)	observableWrite(memory, dst, buf)
read(src, buf)	observableRead(memory, src, buf)

Table 3: Example user-defined functions accessing observable spaces. send and recv are used to express message transfer over network and read and write represents local memory access. network and memory are initialized by users with unique IDs and memory space to store written and sent data.

3.1 ObliCheck APIs

To provide a framework that can accommodate algorithms with different threat models, ObliCheck provides abstract observable and unobservable memory space. Any read and write operations to the observable space are assumed to be observed by an attacker. ObliCheck provides algorithm designers with special APIs for describing reads and writes to the observable space as described in Table 2. We assume data written to or read from observable space is always encrypted. Thus, an attacker can learn the size, source/destination address of the data, and the type of operation (read or write) but not the actual content. Using this abstract store model with APIs, a designer can reflect a threat model that she assumes in the code.

ObliCheck offers two categories of APIs for a designer to write an oblivious algorithm. The first has functions that describe communication between unobservable and observable spaces. The second one is to specify whether an input value is secret or public. Table 2 lists the APIs that ObliCheck provides. Using *observableRead* and *observableWrite*, a designer can naturally render a boundary between observable and observable spaces in the algorithm.

ObliCheck keeps the access sequence under the hood and uses the access sequence to check the final verification condition explained in §3.3. *readSecretInput* and *readPublicInput* let a designer specify the secret input of an algorithm. This specification is necessary to generate the verification condition at the end of symbolic execution. Listing 2 shows the code in Listing 1 re-written using ObliCheck's API.

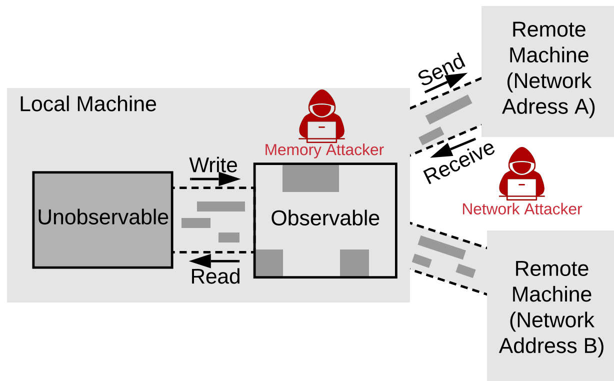


Figure 1: Threat model of ObliCheck. The dark gray part of the figure represents the store and data that an adversary cannot observe. An attacker is not able to eavesdrop the unobservable space and the content of encrypted data. However, an attacker is capable of learning the size of transferred data, the locations of data written to or read from an observable space, and the destination and source network addresses of the network messages and their sizes.

3.2 Threat Model

As discussed in §3.1, ObliCheck assumes an input algorithm leverages unobservable space where an attacker cannot watch the data inside and access patterns over them. ObliCheck considers an attacker watches any accesses to observable space. However, the attacker cannot learn about the actual content of data written to or read from observable space because the data is encrypted when they cross the boundary between unobservable and observable spaces.

This abstract threat model allows algorithm designers to express common threat models that oblivious algorithms assume using the APIs of ObliCheck. For example, the network attacker discussed in §1 can be modeled by using *observableWrite* and *observableRead* for network *send* and *receive* functions respectively. The memory attacker can be modeled in a similar way. Table 3 shows how these functions can be defined using APIs of ObliCheck. We focus on the network adversary as running examples but an algorithm assuming the memory attacker can be checked the same way.

ObliCheck only checks the obliviousness of a given algorithm and assumes the data is properly encrypted when it is written to an observable location. Mistakes of not properly encrypting data can be caught using existing information flow checking techniques [19,23,25,31,40,42,49,56,60,62,67,70,75,84–86]. We assume the code is either inside unobservable space such as oblivious memory pools [20,22,53] or the code accesses are separately treated to be oblivious.

3.3 Security Guarantee

To formulate the security guarantees of ObliCheck, we first define the *trace of observations* visible to the adversary during an execution. Given a algorithm P with input I, the trace of observations τ is defined as a sequence of triplets:

$$\tau_P(I) = < (t_i, a_i, l_i) | i \in N >$$

```
function tag(secretInput, threshold) {
    var buf = [];
    for (var i = 0; i < secretInput.length; i++) {
        if (secretInput[i] < threshold) {
            buf.push(Pair(secretInput[i], 0));
        } else {
            buf.push(Pair(secretInput[i], 1));
        }
    }
    send(ADDR, buf);
}
function main(n) {
    var secretInput = new Array(n);
    for (var i = 0; i < n; i++) { secretInput[i] =
        ObliCheck.readSecretInput() };
    var threshold = ObliCheck.readPublicInput();
    tag(secretInput, threshold);
}
```

Listing 2: Listing 1 is re-written using the APIs of ObliCheck. Only the `socket.send` is replaced with `send`, and the input is introduced using `readSecretInput`, and `readPublicInput`.

where t represents a type of access, a denotes a target or source location of the operation, and l represents the size of a data read or written. The type of access is either read or write combined with the type of an observable space (*e.g.*, memory or network). Further, since we assume the data itself is encrypted properly before being written to an observable store, the attacker can only observe the size of the data that is read or written, and not the actual contents.

Note that in addition to secret data, an algorithm P may also receive some public data as input. For P to achieve the oblivious property, we require that given any pair of inputs I and I', as long as the public input is the same, then no polynomial-time adversary should be able to distinguish between the traces $\tau_P(I)$ and $\tau_P(I')$. Based on this definition, a condition for checking the oblivious property can be expressed as follows:

$$\begin{aligned} &\forall I,\ I' \in InputSpace(P), \\ &\quad PublicInput_P(I) = PublicInput_P(I') \\ &\quad\quad \Rightarrow \tau_P(I) = \tau_P(I') \end{aligned}$$

Here, *InputSpace* represents all the possible input spaces of a given algorithm, and $PublicInput_P$ returns the public input of a algorithm P. ObliCheck verifies that the above condition holds while checking a algorithm. The condition assumes nothing about *SecretInput*, which encodes the independence of the observable output from secret input.

ObliCheck records the trace during the execution under the hood when it encounters a read or write API explained in §3.1. The verification condition is written in terms of the pairs of input (I, I'). This implies that the verification condition for the oblivious property is a 2-safety property [78] that requires a checker to observe two finite traces of an algorithm. We will describe how ObliCheck uses symbolic execution to check the above verification condition in § 4.1.

4 Symbolic Execution and State Merging

4.1 Symbolic Execution for Checking Obliviousness

ObliCheck executes an algorithm symbolically, and at the end of the execution, it checks whether the algorithm satisfies the obliviousness condition defined in §3.3. ObliCheck uses symbolic execution in the following way.

ObliCheck starts by treating all input values as symbolic variables. ObliCheck explores both the true and false blocks of all branches containing a symbolic value, while distinguishing between secret and public symbolic variables to correctly generate the verification condition at the end of the execution.

However, just running an algorithm once symbolically is not sufficient because the verification condition of obliviousness is written in terms of pairs of input. In other words, obliviousness is a 2-safety property. Terauchi and Aiken [78] formally defined a 2-safety property to distinguish it from a general safety property, which can be proved by observing a single finite trace.

In order to refute a 2-safety property, a checker has to observe two finite traces of an algorithm. Hence, ObliCheck internally runs the algorithm twice symbolically, by sequentially composing two copies of the algorithm. Each execution path of the first copy is followed by each one of the second copy. This makes ObliCheck explore every pair (Cartesian product) of the execution paths with pairs of input $(I, I') \in InputSpace(P)$ At the end of the second execution, ObliCheck compares the traces of both runs and checks that the verification condition is always true using a constraint solver (which checks that the negation of the verification condition is unsatisfiable).

Example. To demonstrate how symbolic execution is used, we summarize the result of symbolic execution of Listing 2 in Table 4. For brevity, we assume the input length `n` is 1 so the loop iterates only once. We will generalize for algorithms with loops bounded by an arbitrary symbolic value in § 6.

`main` introduces secret and public symbolic variables x_0 and y respectively and assign them to `secretInput[0]` and `threshold`. To differentiate the first and second symbolic execution, we add additional subscripts $first$ and $second$ to the variables. Inside `tag` function, the first symbolic execution starts with an initial path condition $True$ and the length of the output buffer is 0. After encountering the loop at Line 4, the execution diverges into two sets and the output buffer length increments by one. The second symbolic execution runs the same algorithm but with different symbolic variables: $x_{0,second}$ and y_{second} instead of $x_{0,first}$ and y_{second}.

After finishing the symbolic execution, ObliCheck generates a verification condition based on the definition in §3.3:

Line	Path Condition	buf.length	i	buf[i]
2-4	$\phi_1 = True$	0	0	Undefined
5,8-10	$\phi_1 = x_{0,first} < y_{first}$	1	0	Pair($x_{0,first}$, 0)
7,8-10	$\phi_1 = x_{0,first} \geq y_{first}$	1	0	Pair($x_{0,first}$, 1)
2-4	$\phi_1 = True$	0	0	Undefined
5,8-10	$\phi_1 = x_{0,second} < y_{second}$	1	0	Pair($x_{0,second}$, 0)
7,8-10	$\phi_1 = x_{0,second} \geq y_{second}$	1	0	Pair($x_{0,second}$, 1)

Table 4: Result of symbolic execution of the algorithm in Listing 2.

$$
\begin{aligned}
&y_{first} = y_{second} \Rightarrow \\
&\quad ((x_{0,first} < y_{first} \wedge x_{0,second} < y_{second} \Rightarrow 1 = 1) \\
&\quad \wedge (x_{0,first} < y_{first} \wedge x_{0,second} \geq y_{second} \Rightarrow 1 = 1) \\
&\quad \wedge (x_{0,first} \geq y_{first} \wedge x_{0,second} < y_{second} \Rightarrow 1 = 1) \\
&\quad \wedge (x_{0,first} \geq y_{first} \wedge x_{0,second} \geq y_{second} \Rightarrow 1 = 1))
\end{aligned}
$$

This formula is trivially always true since `buf.length` is always a concrete value 1 (we leave out the type of access and the address fields of the trace for simplicity). The verification condition is quite trivial for this simple example, but as an input algorithm becomes more complicated, symbolic execution proves its real worth since it can capture how the observable trace changes over the execution and can exercise all possible execution paths.

4.2 Optimistic State Merging

Since symbolic execution diverges into two runs when it encounters a branch, the number of executions paths grows exponentially in the number of branches encountered in the execution. This *path explosion* problem inhibits symbolic execution from exploring all possible input space and deteriorates the coverage of a checker as the input length increases. To solve this problem, we devise *optimistic state merging* – a state merging technique that leverages domain-specific knowledge of oblivious execution in the presence of unobservable state.

Shortcomings of Traditional State Merging. Returning to the branch example in Listing 1, the code is oblivious under the definition in §3.3 assuming the data length is public. The algorithm always sends the buffer with a length `n` regardless of the secret values in `secretInputRecords`. To check this condition, a checker should confirm the length of `encrypted` is the same across any possible pairs of `secretInputRecords`. Naïvely running symbolic execution in this example leads to path explosion because the branch is inside the for loop. Since it is common to iterate over elements in the input data set within unobservable space, we need a way to prevent path explosion in this case.

To mitigate the path explosion problem, state merging techniques merge two different symbolic states of a variable. This prevents some unnecessary exploration. However, traditional state merging techniques cannot merge symbolic states when two states are different from each other. For example, Table 4 shows the symbolic states after the execution in Listing 2. With traditional state merging, the `true` and `false` paths of the `if` statement at Line 4 cannot get merged because the `buf[i]` has different state in each path. In other words, traditional state merging techniques are sound and complete with regard to symbolic execution and explores the same set of program behaviors as regular symbolic execution.

In contrast, ObliCheck is able to apply state merging more aggressively through a domain specific insight. Optimistic state merging leverages the observation that, in oblivious algorithms, the attacker is unable to distinguish between different unobservable states because the plaintext data only resides in unobservable space, and is later encrypted when written to observable space. For example, `buf[i]` in Listing 2 is encrypted when the `buf` is sent over network at Line 10. Therefore, at branching statements, ObliCheck explores both true and false blocks immediately and merge the corresponding states into a new symbolic variable without divergence.

Merging Paths by Introducing a New Symbolic Variable. ObliCheck simplifies path conditions by introducing a new variable when merging two different symbolic expressions. For example, the algorithm in Listing 2 exhibits different state of `buf[i]` in the `then` and `else` branches after Line 4 (Pair(x_0,0) and Pair(x_0,1) respectively; Table 4). Hence, traditional state merging cannot merge these two states. In contrast, ObliCheck introduces a new unconstrained symbolic variable, z, and merges the states as in Table 5.

Line	Path Condition	buf.length	i	buf[i]
2-4	$\phi_1 = True$	0	0	Undefined
5	$\phi_1 = x_{0,first} < y_{first}$	1	0	Pair($x_{0,first}$, 0)
7	$\phi_1 = x_{0,first} \geq y_{first}$	1	0	Pair($x_{0,first}$, 1)
8-10	$\phi_1 = True$	1	0	Pair($x_{0,first}$, z)
2-4	$\phi_1 = True$	0	0	Undefined
5	$\phi_1 = x_{0,second} < y_{second}$	1	0	Pair($x_{0,second}$, 0)
7	$\phi_1 = x_{0,second} \geq y_{second}$	1	0	Pair($x_{0,second}$, 1)
8-10	$\phi_1 = True$	1	0	Pair($x_{0,second}$, z)

Table 5: Result of optimistic state merging of the Listing 2.

This merging simplifies the verification condition to $y_{first} = y_{second} \Rightarrow 1 = 1$, which reduces the burden of a constraint solver. Optimistic state merging is an over-approximation based on the domain-specific knowledge of oblivious algorithms, where the data is encrypted and not observable by an adversary. Since it is over-approximation, this a sound transformation; namely, if the transformed symbolic execution judges an algorithm is oblivious, then the original algorithm is always oblivious.

Tracking the Secret Values after Merging. ObliCheck checks the verification after the execution of two copies of a given algorithm. The verification condition in §3.3 is generated from the access sequence recorded by ObliCheck under the hood. To generate the verification condition, ObliCheck needs to know which symbolic values are secret or public.

To this end, ObliCheck associates a taint tag with every introduced symbolic variable. Symbolic variables introduced by *readSecretInput* are assigned a taint tag 1, and the others

$$
\begin{array}{rcl}
Pgm & ::= & (\ell:\ stmt\,;)^* \\
stmt & ::= & x = c \\
 & & x =\ readSecretInput \\
 & & x =\ readPublicInput \\
 & & z = x \bowtie y \\
 & & \text{if } x \text{ goto } y \\
 & & y = *x \\
 & & *x = y \\
 & & \text{error} \\
 & & \text{halt}
\end{array}
$$

where

Σ	is the program state
V	is a set of variables
C	is the set of constants
L	is the set of statement labels
A	is a set of memory addresses
x, y, z	are elements of V
pc	an element of V denoting the program counter
c	is an element of $C \cup A \cup L$
ℓ	is an element of L
$\bowtie$	is a binary operator
SecretSet	is a set of secret symbolic variables
PublicSet	is a set of public symbolic variables

Figure 2: A simple imperative language originally devised by Sen *et al.* in MultiSE [73], augmented with states *SecretSet* and *PublicSet* to maintain the mapping from symbolic values to the taint state. The functions *readSecretInput* and *readPublicInput* introduce a symbolic variable and initialize the corresponding taint tag. Refer to Figure 3 for more details.

are assigned 0. ObliCheck sees the taint tag of symbolic values included in the trace and produces a proper verification condition based on this information. Figure 3 describes the semantics in a formal notation.

The use of taint tag is necessary due to optimistic state merging. When ObliCheck applies optimistic state merging, it has to maintain whether a newly generated symbolic variable is secret. Taint tag lets ObliCheck track how secret input is propagated and decide the security level of a newly generated symbolic variable after optimistic state merging. Unlike traditional taint analysis, ObliCheck draws the final verdict by solving the verification condition not simply from the value of taint tags.

Optimistic State Merging Semantics. Our optimistic state merging technique is based on MultiSE [73]. MultiSE merges state without introducing auxiliary variables, and does not require control flow graph analysis to identify join points because the merging is done incrementally per assignment operation. MultiSE maintains the state of variables in the form of a value summary, a set of path conditions and possible values of a variable. Each pair represents a possible value which a variable can have and the corresponding condition that leads to it. For example, buf.length in Listing 2 can be represented using value summary $\{(x_0 < y, 1), (x_0 \geq y, 1)\}$ after the first loop iteration.

In MultiSE, state merging can be done by simply replac-

Domain Specific Guarded Update

$$\frac{}{\{(\phi_i^a, < v_i^a, t_i^a >)\}_i \mathbin{\cup\!\!\!\!\times}_\phi \{(\phi_j^b, < v_j^b, t_j^b >)\}_j = \{(\neg\phi \wedge \phi_i^a, < v_i^a, t_i^a >)\}_i \mathbin{\cup\!\!\!\!\times} \{(\phi \wedge \phi_j^b, < v_j^b, t_j^b >)\}_j}$$

NextPC

$$\frac{}{\mathit{NextPC}(\Sigma, \phi, \ell) = (\Sigma(\mathit{pc}) \setminus \{(\phi, \ell)\}) \uplus \{(\phi, \ell + 1)\}}$$

Constant

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (x = c)}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \mathbin{\cup\!\!\!\!\times}_\phi \{(\mathit{true}, < c, F >)\}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Symbolic Public Input

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (x = \mathit{readPublicInput}) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \mathbin{\cup\!\!\!\!\times}_\phi \{(\mathit{true}, < s, F >)\}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)][\mathit{PublicSet} \mapsto \Sigma(\mathit{PublicSet}) \cup \{s\}]}$$

Symbolic Secret Input

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (x = \mathit{readSecretInput}) \qquad s \text{ is a fresh symbolic value from } S}{\Sigma \longrightarrow \Sigma[x \mapsto \Sigma(x) \mathbin{\cup\!\!\!\!\times}_\phi \{(\mathit{true}, < s, T >)\}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)][\mathit{SecretSet} \mapsto \Sigma(\mathit{SecretSet}) \cup \{s\}]}$$

Binary Operation

$$\frac{\begin{array}{c}(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (z = x \bowtie y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, t_i^x >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, < v_j^y, t_j^y >)\}_j \\ \phi_{ij}^{x \bowtie y} = \phi_i^x \wedge \phi_j^y \qquad v_{ij}^{x \bowtie y} = v_i^x \bowtie v_j^y \qquad t_{ij}^{x \bowtie y} = t_i^x \vee t_j^y\end{array}}{\Sigma \longrightarrow \Sigma[z \mapsto \Sigma(z) \mathbin{\cup\!\!\!\!\times}_\phi \{(\phi_{ij}^{x \bowtie y}, < v_{ij}^{x \bowtie y}, t_{ij}^{x \bowtie y} >)\}_{ij}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Conditional

$$\frac{\begin{array}{c}(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (\text{ if } x \text{ goto } y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, \ell_j^y)\}_j \\ s = \{(\phi_i^x \wedge v_i^x \wedge \phi_j^y, \ell_j^y)\}_{ij} \uplus \{((\phi_i^x \wedge \neg v_i^x), \ell + 1)\}_i\end{array}}{\Sigma \longrightarrow \Sigma[\mathit{pc} \mapsto (\Sigma(\mathit{pc}) \setminus \{(\phi, \ell)\}) \uplus_\phi s}$$

Load

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (y = *x) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(v_i^x) = \{(\phi_{ij}, < v_{ij}, t_{ij} >)\}_j}{\Sigma \longrightarrow \Sigma[y \mapsto \Sigma(y) \mathbin{\cup\!\!\!\!\times}_\phi \{(\phi_i^x \wedge \phi_{ij}, < v_{ij}, t_{ij} >)\}_{ij}][\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Store

$$\frac{(\phi, \ell) \in \Sigma(\mathit{pc}) \qquad \mathit{Pgm}(\ell) = (*x = y) \qquad \Sigma(x) = \{(\phi_i^x, < v_i^x, \cdot >)\}_i \qquad \Sigma(y) = \{(\phi_j^y, < v_j^y, t_j^y >)\}_j}{\Sigma \longrightarrow \Sigma[v_i^x \mapsto \Sigma(v_i^x) \mathbin{\cup\!\!\!\!\times}_{\phi \wedge \phi_i^x} \{(\phi_j^y, < v_j^y, t_j^y >)\}_j)]_i[\mathit{pc} \mapsto \mathit{NextPC}(\Sigma, \phi, \ell)]}$$

Figure 3: The semantics of symbolic execution and state merging techniques of ObliCheck. The semantics incorporates the taint tag into the MultiSE semantics [73] in order to track the propagation of secret input through merged symbolic values.

ing pairs with the same values with a single pair whose path condition is the disjunction of the conditions of the merged pairs. For instance, buf.length, $\{(x_0 < y, 1), (x_0 \geq y, 1)\}$, becomes $\{(True, 1)\}$ after state merging. MultiSE further removes pairs whose path condition is false when merging.

To formally demonstrate the semantics of ObliCheck operations including optimistic state merging, we bring a simple imperative language from MultiSE [73] in Figure 2. Figure 3 defines the operational semantics of ObliCheck. Each operator updates the program state Σ. The initial state maps each variable to $\{(\mathit{true}, \perp)\}$, and pc to $\{(\mathit{true}, l_0)\}$. To incorporate the taint tag, we extend the value part of the value summary from (ϕ, v) to $(\phi, \langle v, t \rangle)$, where t is the taint tag associated with the value. $\uplus$ is the original value-summary union operator that performs state merging in MultiSE. To distinguish our optimistic state merging operator from the MultiSE operator, we introduce $\mathbin{\cup\!\!\!\!\times}$ operator in the semantics description. Our optimistic state merging operator works as follows.

- In the value-summary pairs, the value part has an additional taint tag t. 1 represents the corresponding value is secret, and 0 denotes the value is public.
- For any two pairs $(\phi, \langle v, t \rangle)$ and $(\phi', \langle v', t' \rangle)$ where $v = v'$, a new value summary for s is calculated in the same way as the $\uplus$ does except that the new taint tag is set to $t \vee t'$. The new value summary becomes $(s \backslash \{(\phi, \langle v, t \rangle), (\phi', \langle v', t' \rangle)\}) \cup \{(\phi \vee \phi', \langle v, t \vee t' \rangle)\}$.
- For any two pairs (ϕ, v) and (ϕ', v') where $v \neq v'$ in a value summary for s, a new symbolic variable y is introduced. If ϕ or ϕ' contain a secret symbolic variable, the new value summary becomes $(s \backslash \{(\phi, \langle v, t \rangle), (\phi', \langle v', t' \rangle)\}) \cup \{(\phi \vee \phi', \langle y, T \rangle)\}$. Otherwise, the value summary becomes $(s \backslash \{(\phi, \langle v, t \rangle), (\phi', \langle v', t' \rangle)\}) \cup \{(\phi \vee \phi', \langle y, t \vee t' \rangle)\}$

For example, buf[i] in the Listing 2 has a value summary $\{(x_0 < y, \langle 0, F \rangle), (x_0 \geq y, \langle 1, F \rangle)\}$. After merging, the new value summary becomes $\{(True, \langle z, T \rangle)\}$. The taint tag after merging is T because the original path conditions contain x_0, a secret symbolic variable even though the original merged values 0 and 1 are not secret values.

The $\mathbin{\cup\!\!\!\!\times}$ operator is used in Figure 3 to describe the semantics of symbolic execution and merging techniques used by ObliCheck. Note that the program counter is treated in the same way as MultiSE using $\uplus$ operator.

5 Iterative State Unmerging

Although our optimistic state merging technique improves the performance of ObliCheck without losing soundness, the overapproximation of the technique incurs false positives. In this section, we point out the problem with optimistic state merging and devise a technique that iteratively and selectively removes false positives.

5.1 Problem of Aggressive State Merging

Optimistic state merging overapproximates the values to get merged. This overapproximation enables more values to be merged but loses path-specific information. Because the values are replaced with symbolic variables which can be an arbitrary value satisfying a corresponding path condition, it brings up more false positives.

Listing 3 is a benign oblivious algorithm but reported as not oblivious if our optimistic state merging is used. At Line 6 and 8, the $i-th$ position of the `buf` is updated to either 0 or 1 depending on the value of the `secretInput[i]`. Since $0 \neq 1$, our optimistic state merging operation introduces a new symbolic variable and put it in the value summary of `buf[i].second`. At Line 16 and 18, the predicates in the branches contain `record.second`, where each `record` points to the value stored at `buf[i]`. Since ObliCheck overapproximated the `buf[i].second`, it has no way to know 0 and 1 are the only possible values for `record.second` and thus the algorithm is reported as not oblivious.

```
1  function tag(secretInput, threshold) {
2      var buf = [];
3      for (var i = 0; i < secretInput.length; i++) {
4          if (secretInput[i] < threshold)
5              buf.push(Pair(secretInput[i], 0));
6          else
7              buf.push(Pair(secretInput[i], 1));
8      }
9      return buf;
10 }
11 function apply(records, func0, func1) {
12     var buf = [];
13     for (var i = 0; i < records.length; i++) {
14         if (records[i].second == 0)
15             buf.push(func0(records[i].first));
16         if (records[i].second == 1)
17             buf.push(func1(records[i].first));
18     }
19     return buf;
20 }
21 function main() {
22     // Input values are initialized
23     ...
24     var tagged = tag(secretInput, publicThreshold);
25     ...
26     var applied = apply(tagged, funcA, funcB);
27     ...
28     applied = Cipher.encrypt(applied);
29     write(ADDR, applied);
30 }
```

Listing 3: Example code from Opaque [89]. `tag` function tag 0 or 1 depending on the value of each `secretInput[i]` of `secretInput`. `apply` applies a function to the value depending on the tag of an element. Optimistic state merging merges the tags 0 and 1 into a symbolic value. Although the branches in `apply` do not cause non-oblivious behavior, the algorithm is reported as non-oblivious because the `record.second` becomes a symbolic value after merging.

Our merging technique does not affect the soundness of ObliCheck, but sacrifices the completeness due to the overapproximation for merging. In fact, if we merge every variable, any algorithms that have a secret dependent branch that affects the access sequence are classified as not oblivious, the same way as a taint analysis based checker does. For better precision, ObliCheck has to intelligently choose variables to apply the optimistic state merging technique.

5.2 Iteratively and Selectively Unmerging State

To overcome the above issue, we introduce an iterative way to remove false positives. Choosing which values to merge during the execution is tricky. The symbolic execution engine does not immediately know how an updated variable is used later by the verification condition. Simply rolling back the merged state after the symbolic execution significantly deteriorates the performance of ObliCheck when a given algorithm is a false-positive, where the OSM classifies the algorithm as not oblivious but it is actually oblivious.

Instead of identifying which variables to merge, ObliCheck does the reverse. ObliCheck first runs a program merging every variable updated in multiple execution paths. Then it checks the verification condition, and identifies which variables should be unmerged. In the next iteration, ObliCheck backtracks the execution, locates operations where the merging should be avoided and re-runs the program symbolically. The verification is performed again at the end of the iteration. This iterative process helps ObliCheck learn how a certain merging operation affects the outcome of verification later.

Algorithm 2 in Figure 5 is a formal description of the iterative state unmerging process. During the execution, ObliCheck tracks the location of operations which incur the domain-specific merging. Jalangi inserts a unique operation ID for every operation in a program statically. ObliCheck stores the ID of operations which introduced a symbolic variable or triggered domain-specific merging to an introduced symbolic variable. At the end of each iteration, symbolic variables included in the verification condition are extracted. If the verification condition does not hold and the extracted symbolic variables contain ones introduced by domain-specific merging, the operation IDs stored in *SymVarToOID* are added to *UnmergeOID* to prohibit merging at these locations in the next iteration. This iterative process enables an efficient selection of merging points that do not incur false positive error.

An algorithm with more non-oblivious branches will end up enduring more unnecessary iterations, wasting time. However,

Algorithm 1 Iterative state unmerging algorithm

```
global variables
    SymVarToOID                ▷ Symbolic variables to operation IDs
    UnmergeOID                 ▷ Set of operation IDs
end global variables
  ▷ Called for every assignment operation in a program
procedure UPDATE(OperationID)
    if OperationID ∈ UnmergeOID then
        ConventionalMerging(OperationID)
    else
        s ← DomainSpecificMerging(OperationID)
        SymVarToOID[s] ← SymVarToOID[s] ∪
                    {OperationID}
procedure OBLICHECKMAIN(Program)
    while true do
        Reset SymVarToOID
        Trace1 ← SymbolicExec(Program)
        Trace2 ← SymbolicExec(Program)
        VC ← ObliviousVC(Trace1, Trace2)
        if VC then
            report OBLIVIOUS, break
        else
            SymVarsInVC ← ExtractSymVars(VC)
            if SymVarsInVC ∩ SymVarToOID.keys ≠ ∅ then
                for all s ∈ SymVarsInVC do
                    UnmergeOID ← UnmergeOID ∪
                            SymVarToOID[s]
            else
                report NOT OBLIVIOUS, break
```

Figure 4: A formal description of how our iterative state unmerging algorithm functions. *SymVarToOID* is a dictionary maps a symbolic variable introduced by merging to a set of operation IDs. The operation IDs uniquely identify each operation in a program statically. *UnmergeOID* is a set of operation IDs that represent the locations where ObliCheck should avoid performing our domain-specific merging. For every iteration, *UnmergeOIDs* grows. This lets ObliCheck increases the precision gradually as necessary.

our domain-specific merging was based on the expectation that developers checking a algorithm for obliviousness likely put effort towards making it oblivious, potentially missing a few details. Therefore, the number of iterations required to unmerge relevant symbolic values is not large. In §7, we evaluate the additional cost using example algorithms. If ObliCheck fails to check a algorithm within a given time budget, it reports the locations where state merging has happened. This information can greatly assist a algorithm designer to manually inspect only a part of the code and then figure out whether the algorithm is a true-positive or false-positive.

6 Handling Input-dependent Loops

6.1 Problem of Loops Bounded by Symbolic Expression

A well-known limitation of symbolic execution is its inability of verifying a program containing an input-dependent loop. These types of loops are bounded by a symbolic expression which consists of input symbolic variables. A pro-

```
// threshold and inputSize are public input
function tag(secretInput, threshold, inputSize) {
    var buf = [], i = 0;
    while (i < inputSize) {
        if (secretInput[i] < threshold) {
            // buf.length += 1 inside push
            buf.push(Pair(secretInput[i], 0));
        } else {
            // buf.length += 1 inside push
            buf.push(Pair(secretInput[i], 1));
        }
        i++;
    }
    return buf;
}
```

Listing 4: `tag` function with an input-dependent loop. The `for` loop is transformed into `while` to better demonstrate the control flow.

gram containing an input-dependent loop has an infinite number of paths for a symbolic execution engine to explore. For example, Listing 4 shows a loop bounded by `inputSize`. The path condition of the first iteration inside the loop is $0 <$ `inputSize`. That of the second one is $\neg(0 <$ `inputSize`$) \wedge (1 <$ `inputSize`$)$ and a new path condition is generated infinitely since `inputSize` is not bounded.

Most oblivious algorithms involve loops bounded by input symbolic variables. These loops are used to iterate over an input secret record of which the length is public. The length of the processed output is thus dependent on the input length. However, the algorithm can still be oblivious since revealing the input length does not violate the obliviousness property. In order to verify generalized oblivious algorithms with symbolic input length, ObliCheck is required to handle loops bounded by symbolic variables.

6.2 Automatic Generation of Loop Invariants

In a general program verification, a user is required to provide a loop invariant manually since it is an undecidable problem [12, 29, 41, 51, 76]. However, ObliCheck automatically infers relevant partial loop invariants by leveraging a fact that the length of the output is an *induction variable*. Induction variables get incremented or decremented by a fixed amount for each iteration in a loop. Oblivious algorithms use input-dependent loops to build up output data by iterating over the secret input records. To preserve the obliviousness, a fixed amount of elements are appended to the output buffer for every iteration as shown in the tagging example of Listing 4.

As long as the size of a buffer is an induction variable, the problem is reduced to inferring the number of iterations of a loop. The side-effects of a loop to induction variables can be captured by multiplying the delta of the variables per iteration by the number of iterations. Godefroid and Luchaup [35] formalized this idea in dynamic test generation which produces test inputs while executing the program concretely. We extend the idea to capture partial loop invariants in pure symbolic execution. In a similar way that Godefroid and Luchaup [35]

Algorithm 2 Automatic loop invariant generation algorithm

```
    ▷ Called for every read operation in a loop
procedure READLOOP(L, Var)
    if Var not in L.UpdatedVars.Keys then
        L.UpdatedVars[Var] = readSecretInput
    return L.UpdatedVars[Var]
    ▷ Called for every write operation in a loop
procedure UPDATELOOP(L, Var, Val)
    L.UpdatedVars[Var] = Val
    ▷ Both functions are called at the end of a loop body
procedure INFERINDUCTIONVARS(L)
    for V in L.UpdatedVars.Keys do
        if L.Iteration == 1 then
            L.IVCandidates[V]=L.UpdatedVars[V]
        if L.Iteration == 2 then
            L.IVDeltas[V]=L.UpdatedVars[V]-L.IVCandidates[V]
            L.IVCandidates[V]=L.UpdatedVars[V]
        if L.Iteration == 3 then
            if L.UpdatedVars[V] - L.IVCandidates[V]
               == L.IVDeltas[V] then
                IVs.append(V)
    return IVs
procedure INFERLOOPITERATIONS(L)
    for C in L.LoopConditions do
        if L.Iteration == 1 then
            C.Value = C.LHS - C.RHS
        if L.Iteration == 2 then
            C.Delta = (C.LHS - C.RHS) - C.Value
        if L.Iteration == 2 then
            if (C.LHS - C.RHS) - C.Value == C.Delta then
                if L.Operator == < then
                    C.LoopCount = -(C.InitialVal / C.Delta)
                if L.Operator == > then
                    ...
```

Figure 5: Functions added for generating loop invariants automatically. ReadLoop and UpdateLoop track the changed variables inside the loop. ReadLoop returns a fresh symbolic variable if a variable is read before written. InferInductionVars and InferLoopIterations track the delta of the variables and loop conditions to find the induction variables, and compute the number of iterations of a loop.

proposed, ObliCheck tracks the modified variables and check the delta of the variables and expression in the loop condition between two consecutive iterations. Unlike Godefroid and Luchaup, however, we use pure symbolic execution for sound verification and finish loop summarization within three iterations by over-approximation.

Finding Induction variables. ObliCheck figures out the difference of each variable between the first and second iterations, and the second and third ones. Then ObliCheck checks that the two differences are the same. The first iteration starts with an empty state mapping. When a variable is modified in the first iteration, an entry from the variable to its concrete or symbolic value is updated. If a variable is referenced but it does not have an entry in the mapping, an unconstrained symbolic variable is assigned to the referenced variable. This over-approximation takes any possible modifications in previous iterations into account. At the end of the first iteration, the values of the updated variables are saved. The second iteration is executed with the state created during the first iteration. At the end of the second iteration, the difference of the values saved at the first iteration and the second one is calculated and saved. After the third iteration, another set of the deltas is obtained and the variables whose deltas are the same are judged as induction variables.

Calculating the number of iterations. The number of loop iterations depends on the loop condition that bounds the loop. Loop conditions are the conditional statements inside a loop that have one of their targets point to the outside of the loop. A conditional predicate of the form $LHS \circ RHS$ in a loop condition, where $\circ$ is one of the conditional operators ($<, \leq, >, \geq, =, \neq$), can be transformed to $LHS - RHS \circ 0$ and the delta of $LHS - RHS$ between iterations are obtained in the same way that the delta of induction variables are figured out [35]. When the operator $\circ$ is $<$, the number of iterations is $-(InitialValue/Delta)$. Since there can be multiple loop conditions if a loop body has `break` or `return` statement, ObliCheck computes the number of iterations for each loop condition and takes the minimum among them.

After getting the delta per iteration of induction variables and the number of iterations, the loop's post-condition becomes $\bigwedge_i^n IV_i = C_i + D_i * IC_l$, where IV_i represents the induction variables, C_i is each induction variable's initial value before the loop, and IC_l is the number of iterations of the loop l. For example, the algorithm in Listing 4 has two induction variables, `i` and `buf.length`. The post-condition becomes $i = 0 + 1 * inputSize \wedge buf.length = 0 + 1 * inputSize$. The pre-condition of the loop is the loop condition $i < inputSize$, so the loop is summarized as $(i < inputSize) \wedge (i = inputSize \wedge buf.length = inputSize)$.

Limitation. ObliCheck cannot summarize the side-effects of a loop on non-induction variables. Also, if the loop condition depends on a non-induction variable, ObliCheck is unable to infer the number of loop iterations. In these cases, ObliCheck simply assigns an arbitrary symbolic variable to non-induction variables and variables changed in a loop bounded by non-induction variables for over-approximation. If a part of the over-approximated variables is included in the verification condition, it will result in a false-positive. However, in §7 we show that this is not the case for existing oblivious algorithms since the relevant variables such as the length of the output buffer increment by a fixed amount per iteration.

7 Evaluation

We implemented our checker based on Jalangi [71], a program analysis framework for JavaScript. The choice of Javascript is irrelevant to the core techniques of ObliCheck and the same techniques can be implemented in any programming languages. Jalangi implements symbolic execution dynamically through source code instrumentation. In contrast to static analysis frameworks, Jalangi can avoid imprecise alias or pointer analyses since it actually runs a program under the hood and the input values are restricted to values not references.

We measured the total analysis time including the symbolic execution and constraint solving time, but excluded the instrumentation time which is syntax-based and done before the symbolic execution. The experiment was done on a Linux machine with Ubuntu 18.04.2, Intel Core i7 quad-core CPU and 32 GB of RAM.

We evaluate ObliCheck using existing data processing algorithms from data processing frameworks used in production and published academic papers. Table 6 lists the benchmark algorithms. Opaque [89] is an open-source, distributed data analytics frameworks based on Apache Spark [2]. Signal Messenger [6] is an open-source encrypted messaging service commercialized by Signal Messenger LLC.

Algorithm	Description
Tag	The algorithm in Listing 1
Tag (Not Oblivious)	The algorithm in Listing 1 with the false branch in the if statement removed
Tag&Apply	The algorithm in Listing 3
Sort	Oblivious operator from Opaque [89]
Filter	Oblivious operator from Opaque [89]
Aggregate	Oblivious operator from Opaque [89]
Join	Oblivious operator from Opaque [89]
MapReduce	Oblivious MapReduce by Ohrimenko *et al*. [64]
Decision Tree	Oblivious decision tree inference by Ohrimenko *et al*. [65]
Hash Table	Oblivious hash table used in the Signal messaging service [6]
AES Encryption	AES CBC encryption from AES-JS [1]
Neural Net Inference	Prediction part of a neural network from neuroJS [5]
TextSecure Server	End-to-End message encryption server in Javascript [4]

Table 6: List of benchmark algorithms. Tag and Tag&Apply are the example algorithms showed earlier. Sort, Filter, Aggreate and Join are from the Opaque framework [3], MapReduce and Decision Tree are from Ohrimenko *et al*. [64,65] and Hash Table is from the Signal Messenger [6].

7.1 Accuracy Test

We first evaluate the accuracy of ObliCheck's techniques (i.e., optimistic state merging and iterative state unmerging) and compare it with other existing techniques – namely, taint tracking, and symbolic execution with conventional state merging (MultiSE). Table 7 displays the results. MapReduce is not oblivious because it pads the output up to the possible maximum length of the output based on the input data. Thus, it leaks information regarding the input data distribution. TextSecure Server is not oblivious since the server sends the different length of the message based on the status of the devices and it does not pad the messages before sending them.

Example	Oblivious?	Taint Analysis	ObliCheck	
			OSM	OSM+ISU
Tag	◯	× ✗	◯ ✓	◯ ✓
Tag (NO)	×	× ✓	× ✓	× ✓
Tag&Apply	◯	× ✗	× ✗	◯ ✓
Sort	◯	× ✗	◯ ✓	◯ ✓
Filter	◯	× ✗	◯ ✓	◯ ✓
Aggregate	◯	× ✗	◯ ✓	◯ ✓
Join	◯	× ✗	◯ ✓	◯ ✓
MapReduce	×	× ✓	× ✓	× ✓
DecisionTree	◯	× ✗	◯ ✓	◯ ✓
HashTable	◯	× ✗	◯ ✓	◯ ✓
AES Encryption	◯	◯ ✓	◯ ✓	◯ ✓
Neural Net Inference	◯	◯ ✓	◯ ✓	◯ ✓
TextSecure Server	×	× ✓	× ✓	× ✓

Table 7: Accuracy evaluation result of each technique over the benchmark suite algorithms. Taint Analysis checks the algorithm has a secret dependent branch by taint tracking. OSM is our optimistic state merging technique where only the `length` of buffers are not merged, and ISU is our iterative state unmerging technique (ObliCheck). ◯ means the algorithm is classified as oblivious and × represents one is classified as not oblivious. ✓ marks the test result is correct (either true positive or true negative)and ✗ marks the result is an error (either false positive or false negative).

Taint analysis classifies all algorithms as not oblivious except for AES Encryption and Neural Net Inference. Both of the two are only algorithms without secret-dependent branches. Our optimistic state merging technique obtains the correct results except for the Tag&Apply example, where merging the tag values leads to false positive. Both conventional state merging and our iterative state unmerging technique correctly identify oblivious and non-oblivious algorithms.

7.2 Performance Evaluation

Pure symbolic execution suffers from path explosion and conventional state merging does not fully address this issue. We evaluate the performance of applying conventional state merging to ObliCheck and show how much performance improvement it achieves in terms of total program analysis time. We also measured the overhead of iterative state merging compared with a non-iterative domain-specific merging technique. The length of the input data is 40 except for the AES Encryption and TextSecure Server. AES Encryption requires the number of input bytes is multiple of 16 bytes, so we set the length at 4096. Neural Net Inference runs out of memory at the length of 40 so we set it at 20. The input data to be processed is considered as private in all the examples. In the Neural Net Inference, we consider the size of the network layers is not private. In the TextSecure Server, we consider the destination device addresses are private input.

Table 8 shows the evaluation results of pure MultiSE and

Example	LoC	Symbolic Execution (MultiSE)		ObliCheck (OSM)		ObliCheck (OSM + ISU)			
		Total Time (s)	Avg Value Summary Size	Total Time (s)	Avg Value Summary Size	Total Time (s)	Avg Value Summary Size	Speed Up (×) (vs MultiSE)	Overhead (%) (vs OSM)
Tag	43	2751.61	24.14	1.28	1.41	1.29	1.41	2134.68	0.00
Tag (NO)	44	2765.46	23.19	92.76	2.04	96.75	2.04	28.58	4.03
Tag&Apply	48	3227.15	19.68	0.98	1.44	1.66	1.43	1938.47	69.28
Sort	152	3820.16	8.17	1.05	1.03	1.06	1.03	3592.38	0.61
Filter	162	4272.28	12.22	1.20	1.03	1.21	1.03	3517.08	0.48
Aggregate	183	5573.17	12.73	1.39	1.03	1.39	1.03	4011.28	0.00
Join	183	5285.54	12.73	1.31	1.03	1.32	1.03	3991.17	0.90
MapReduce	76	5384.13	62.32	284.20	1.87	442.33	1.92	12.17	55.64
DecisionTree	61	7506.62	70.79	1.52	1.02	1.55	1.02	4850.34	1.64
HashTable	68	6530.72	38.64	1.66	1.46	1.67	1.46	3912.44	0.67
AES Encryption	797	0.91	1	0.95	1	0.95	1	0.95	0.0
Neural Net Inference	219	6.77	1	6.96	1	6.96	1	0.97	0.0
TextSecure Server	184	17935.37	53.40	84.04	1.35	89.51	1.35	200.37	6.11

Table 8: Performance evaluation result of each technique on the test algorithms. OSM refers to optimistic state merging, and ISU to iterative state unmerging. The total time includes the execution time of the symbolic execution engine and the solver time of ObliCheck. The average value summary size is the average length of the value summary, which reflects how efficiently state merging was done. OSM shows the best performance since it merges everything and executes a program only once. ObliCheck with ISU has less than 5.0% of the overhead for the test algorithms except for Tag&Apply and MapReduce. Two algorithms are a false positive and a true negative, which make ObliCheck iterates more.

ObliCheck on the test algorithms. ObliCheck performs up to ×4850 faster than MultiSE. The improvement mainly comes from the reduced number of exploration paths and simplified path conditions due to optimistic state merging. The overhead of iterative state merging is marginal if the algorithm is oblivious as it iterates only once. If the algorithm is not oblivious (true positive) or needs more iterations to turn out to be oblivious (false positive) the overhead becomes more significant. In the benchmark suite, the maximum overhead is $\sim 69\%$.

We also demonstrate the scalability of ObliCheck compared with conventional state merging techniques, by running vanilla MultiSE and ObliCheck over Tag, Tag&Apply and Non-oblivious Tag algorithms. The algorithms result in a true negative, false positive and true positive respectively when checked using optimistic state merging.

Figure 6 shows the results. ObliCheck boasts linear scalability when it checks Tag, and Tag&Apply algorithms, which are oblivious cases. In contrast, the runtime of MultiSE grows exponentially. For Non-oblivious Tag, the total analysis time of ObliCheck also grows exponentially since it fails to merge the states in the end. In this case, ObliCheck provides the information regarding the program statements where state unmerging has been applied so that a algorithm designer can manually inspect and judge a given algorithm is truly non-oblivious.

Table 9 demonstrates the loop summarization performance of ObliCheck. The number of loops only include ones summarized by ObliCheck. For example, AES Encryption algorithm contains multiple for loops but only one outermost loop has the input length in its loop condition. All the other loops are constants. As we discussed in § 4.1, MultiSE runs infinitely when a given algorithm contains input-dependent loops so cannot verify it. In contrast, ObliCheck generate loop invariants automatically and classifies every test algorithm correctly within a second.

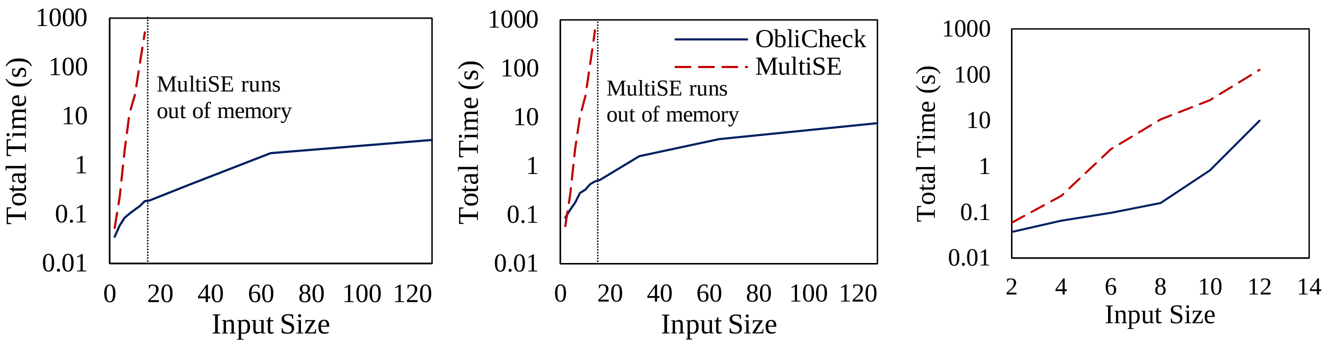
Example	MultiSE	ObliCheck	# of Loops	Total Time (s)
Tag	∞	◯ ✓	1	0.170
Tag (NO)	∞	× ✓	1	0.290
Tag&Apply	∞	◯ ✓	1	0.180
Sort	∞	◯ ✓	20	0.423
Filter	∞	◯ ✓	25	0.477
Aggregate	∞	◯ ✓	31	0.623
Join	∞	◯ ✓	27	0.468
MapReduce	∞	× ✓	8	0.172
DecisionTree	∞	◯ ✓	4	0.062
HashTable	∞	◯ ✓	4	0.074
AES Encryption	∞	◯ ✓	1	0.189
Neural Net Inference	∞	◯ ✓	1	0.296
TextSecure Server	∞	× ✓	1	0.211

Table 9: Loop invariant generation test result. The # of Loops column includes the number of loops summarized by ObliCheck. ∞ means the checking process runs infinitely. MultiSE runs infinitely for all test algorithms because of input-dependent loops. ObliCheck classifies each algorithm correctly by summarizing the loops.

8 Discussion

8.1 Generalization for Checking Other Side Channels

ObliCheck proves the absence of the access pattern side-channel by keeping the access sequence as a program state. Based on the recorded state, ObliCheck checks whether the predefined verification holds at the end of symbolic execution. In principle, other types of side-channel leakage can also be verified in a similar way. For example, one can model timing side-channels by recording the number of steps of a algorithm while symbolically executing a algorithm. In contrast to existing works that rule out algorithms with secret dependent branches and memory accesses entirely [16, 81], comparing the time it takes to finish each execution path directly is a more precise approach. By (1) modeling observable behavior of an algorithm as program state during the symbolic execu-



(a) Oblivious Tagging (True Negative) **(b)** Oblivious Tag and Apply (False Positive) **(c)** Non-oblivious Tagging (True Positive)

Figure 6: Total analysis time of MultiSE (conventional state merging) and ObliCheck (domain-specific merging followed by iterative state unmerging) over Tag, Tag&Apply, and Tag (Non-oblivious). The total time of MultiSE grows exponentially until the input size 16 and fails to finish due to out of memory error after then when it analyzes Tag and Tag&Apply. The total analysis time of ObliCheck grows linearly without out of memory error. The total time of ObliCheck blows up exponentially when it checks the non-oblivious Tag algorithm. This is because state merging is not possible after unmerging merged state and the size of state exponentially grows as MultiSE does.

tion, and (2) defining the verification condition based on the state, one can prove the side-channel leakage using the same technique used in ObliCheck. We leave the generalization of our technique for different types of side-channels as future work.

8.2 Checking Probabilistically Defined Obliviousness

ObliCheck checks if a given algorithm has the same deterministic access sequence across all possible input. In contrast, the original ORAM work defines obliviousness probabilistically. In order to verify the obliviousness condition in this case, a checker should keep the probability distribution of access sequences and verify the distributions of any two inputs are indistinguishable. For this, a symbolic execution engine should be able to capture how a variable with probability distribution is transformed over the algorithm execution. Several works have been proposed recently to automatically verify differential privacy, which certifies the distance between any two algorithm outputs is within a concrete bound [9, 14, 87]. For example, LightDP [87] provides a language with a lightweight dependent type incorporating probability distribution. Similarly, ObliCheck can be extended with APIs or with a new domain-specific language to capture probability distribution, its transformation during the execution. The final verification condition checks the statistical distance of the observable state for any two inputs. This interesting direction requires further investigation and we leave it for future work.

9 Related Work

Checking Side Channel Leakage Using Taint Analysis. Several works have been proposed to detect or mitigate side-channel leakage of an algorithm using taint analysis. Vale [16] provides a domain-specific language (DSL) and tools for writing high-performance assembly code for cryptographic primitives. Vale includes a checker which uses taint analysis that checks the written code is free from digital side-channels of memory and timing. As described in §2.3, this approach can result in a large number of false positives in the presence of unobservable state.

Raccoon [68] uses taint analysis to identify secret dependent branches which can potentially leak information and obfuscate the behaviors of these branches. Since Raccoon is a compiler but not a checker, using taint analysis in this way may result in unnecessary obfuscation but not the rejection of a program. Sidebuster [88] uses taint analysis in the same way to check and mitigate side-channels in web applications. Overall, taint analysis is an efficient technique to detect and mitigate side-channels under a limited time budget. However, it keeps a coarse-grained state regarding information flow in that it only tracks which variables are affected by a source input.

Symbolic Execution and State Merging Techniques for Preventing Side Channel Attacks. Symbolic execution has widely deployed to check certain properties of a program and to generate high-coverage test cases [17, 18, 21, 34, 47, 71, 72]. Practical symbolic execution frameworks normally limit the depth of exploration or drive the execution to parts of a code to find buggy code with a limited time budget. Our checker rather checks the whole input space of a program to eliminate false-negative cases to make our checker useful for checking the security property. Jalangi [71] is a program analysis framework for Javascript where ObliCheck is built atop. Since Jalangi is a dynamic framework, ObliCheck can elude imprecise alias analyses.

State merging techniques are used to resolve the path explosion problem of symbolic execution at the expense of more complicated path conditions [10, 32, 33, 36]. MultiSE [73] merge states incrementally at every assignment of symbolic variables without introducing auxiliary variables. MultiSE supports merging values not supported by constraint solver such as functions and makes it unnecessary to identify the join points of branches to merge state. OSM of ObliCheck is fundamentally different from existing state merging techniques.

Existing state merging techniques are sound and complete with regard to symbolic execution. The merged symbolic state explores the same set of program behaviors as regular symbolic execution. Therefore, existing techniques do not report false positives. In contrast, OSM leverages domain-specific knowledge from oblivious programs and over-approximates program behavior to merge two states even if they cannot be merged in original state merging, which significantly speeds up the checking process. However, OSM might report false positives, and that's where ISU kicks in to repair them.

One of the most widely exploited and studied side-channels is the cache side-channel. CaSym [52] uses symbolic execution to detect a part of a given program that incurs cache side-channel leakage. CaSym runs the LLVM IR of a program symbolically and finds inputs which let an attacker distinguish observable cache state. CaSym merges paths by introducing an auxiliary logical variable. CaSym and our checker use similar symbolic execution techniques but for different purposes. CaSym specifically focuses on checking cache side-channel leakage with a comprehensive cache model but ObliCheck is for more general oblivious algorithms. CacheD [81] also uses symbolic execution but only checks the traces explored in a dynamic execution of a program, which may miss potential vulnerabilities incurred by secret dependent branches. CacheAudit [26] uses abstract interpretation to detect cache side-channel leakage.

Ensuring Noninterference Policy. Noninterference is a security policy model which strictly enforces information with a 'high' label does not interfere with information with a 'low' label [25]. Some existing approaches for enforcing noninterference are type checking [60, 61, 67, 80] and abstract interpretation [8, 30, 48].

Barthe *et al*. defined a way to prove noninterference by a sequential composition of a given algorithm [13]. Terauchi and Aiken proposed a term 2-safety to distinguish safety property like noninterference which requires to observe two finite sets of traces [78]. Also, they devised a type-based transformation of a given algorithm for self-composition which has better efficiency than a simple sequential-composition suggested by Barthe *et al*. for removing redundant and duplicated execution. Milushev *et al*. suggested a way to use symbolic execution to prove the noninterference property of a given algorithm [58]. They used type-directed transformation suggested by Terauchi and Aiken to interleave two sets of algorithms. The type-directed transformation can be orthogonally applied and potentially improve the performance of ObliCheck.

10 Conclusion

Access pattern based side-channels have gained attraction due to a large amount of information it leaks. Although oblivious algorithms have been devised to close this side-channel, the correctness of the algorithms must be manually checked by understanding pen and pencil proofs. We showed that symbolic execution can be utilized to automatically check a given algorithm is oblivious. With our optimistic state merging and iterative state unmerging techniques, ObliCheck achieves more accurate results than existing taint analysis based techniques and runs faster than traditional symbolic execution.

References

[1] Aes-js: A pure javascript implementation of the aes block cipher algorithm and all common modes of operation. `https://github.com/ricmoo/aes-js`. Accessed: 2020-08-12.

[2] Apache spark: Lightning-fast unified analytics engine. `https://spark.apache.org/`. Accessed: 2020-08-10.

[3] Github: Opaue. `https://github.com/ucbrise/opaque`. Accessed: 2020-02-10.

[4] A javascript implementation of a textsecure server. `https://github.com/joebandenburg/textsecure-server-node`. Accessed: 2020-08-12.

[5] neurojs: Neural network library. `https://github.com/pieteradejong/neuroJS`. Accessed: 2020-08-12.

[6] Technology preview: Private contact discovery for signal. `https://signal.org/blog/private-contact-discovery/`. Accessed: 2019-05-06.

[7] An off-chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, August 2020. USENIX Association.

[8] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Abstract certification of global non-interference in rewriting logic. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects*, FMCO'09, pages 105–124, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Aws Albarghouthi and Justin Hsu. Synthesizing coupling proofs of differential privacy. *Proc. ACM Program. Lang.*, 2(POPL):58:1–58:30, December 2017.

[10] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 367–381, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. Secure database-as-a-service with cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1033–1036, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

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

[14] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Advanced probabilistic couplings for differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 55–67, New York, NY, USA, 2016. ACM.

[15] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 441–459, New York, NY, USA, 2017. Association for Computing Machinery.

[16] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. Vale: Verifying high-performance cryptographic assembly code. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 917–934, Vancouver, BC, 2017. USENIX Association.

[17] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[18] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[19] Lorenzo Cavallaro and R. Sekar. Taint-enhanced anomaly detection. In *Proceedings of the 7th International Conference on Information Systems Security*, ICISS'11, page 160–174, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71, May 2016.

[21] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[22] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 857–874, Berkeley, CA, USA, 2016. USENIX Association.

[23] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of tls 1.3. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1773–1788, New York, NY, USA, 2017. Association for Computing Machinery.

[24] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. Privacy-preserving computation with trusted computing via scramble-then-compute. *Proceedings on Privacy Enhancing Technologies*, 2017(3):21–38, 2017.

[25] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[26] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan 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)*, pages 431–446, Washington, D.C., 2013. USENIX.

[27] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '10, pages 13–22, New York, NY, USA, 2010. ACM.

[28] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 693–707, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Robert W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[30] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197, New York, NY, USA, 2004. ACM.

[31] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Confer-*

ence on Operating Systems Design and Implementation, OSDI'12, pages 47–60, Berkeley, CA, USA, 2012. USENIX Association.

[32] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[33] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 47–54, New York, NY, USA, 2007. Association for Computing Machinery.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[35] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 23–33, New York, NY, USA, 2011. Association for Computing Machinery.

[36] Patrice Godefroid, Aditya Nori, Sriram Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Principles of Programming Languages (POPL)*. Association for Computing Machinery, Inc., January 2010.

[37] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, May 1996.

[38] Michael T. Goodrich, Olga Ohrimenko, and Roberto Tamassia. Data-oblivious graph drawing model and algorithms. *CoRR*, abs/1209.0756, 2012.

[39] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Jean-Louis Lanet, and Routa Moussaileb. Detection of side channel attacks based on data tainting in android systems. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 205–218. Springer, 2017.

[40] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R.B. Butler. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2245–2262, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[42] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[43] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *CoRR*, abs/1803.05961, 2018.

[44] Nick Hynes, Raymond Cheng, and Dawn Song. Efficient deep learning on multi-source private data. *CoRR*, abs/1807.06689, 2018.

[45] L. Ivanov and R. Nunna. Modeling and verification of cache coherence protocols. In *ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No.01CH37196)*, volume 5, pages 129–132 vol. 5, May 2001.

[46] S. Kadloor, X. Gong, N. Kiyavash, T. Tezcan, and N. Borisov. Low-cost side channel remote traffic analysis attack in packet networks. In *2010 IEEE International Conference on Communications*, pages 1–5, May 2010.

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

[48] Máté Kovács, Helmut Seidl, and Bernd Finkbeiner. Relational abstract interpretation for the verification of 2-hypersafety properties. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 211–222, New York, NY, USA, 2013. ACM.

[49] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[50] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association.

[51] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, page 348–370, Berlin, Heidelberg, 2010. Springer-Verlag.

[52] X. Ling, S. Ji, J. Zou, J. Wang, C. Wu, B. Li, and T. Wang. Deepsec: A uniform platform for security analysis of deep learning model. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[53] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. Ghostrider: A hardware-software system for memory trace oblivious compu-

tation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 87–101, New York, NY, USA, 2015. ACM.

[54] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[55] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 619–631, New York, NY, USA, 2017. Association for Computing Machinery.

[56] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation. In *MobiSys 2017 - Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys 2017 - Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, pages 225–238. Association for Computing Machinery, Inc, 6 2017.

[57] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. HASP, 2013.

[58] Dimiter Milushev, Wim Beck, and Dave Clarke. Non-interference via symbolic execution. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32Nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 152–168, Berlin, Heidelberg, 2012. Springer-Verlag.

[59] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, May 2018.

[60] Andrew C. Myers and Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[61] Andrew C. Myers, Andrew C. Myers, and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 129–142, New York, NY, USA, 1997. ACM.

[62] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 165–179, Washington, DC, USA, 2011. IEEE Computer Society.

[63] Valeria Nikolaenko, Stratis Ioannidis, Udi Weinsberg, Marc Joye, Nina Taft, and Dan Boneh. Privacy-preserving matrix factorization. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 801–812, New York, NY, USA, 2013. ACM.

[64] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in mapreduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1570–1581, New York, NY, USA, 2015. ACM.

[65] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, 2016. USENIX Association.

[66] Sergei Popov, Stanislav Morozov, and Artem Babenko. Neural oblivious decision ensembles for deep learning on tabular data. *arXiv preprint arXiv:1909.06312*, 2019.

[67] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, January 2003.

[68] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 431–446, Washington, D.C., 2015. USENIX Association.

[69] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. Zerotrace: Oblivious memory primitives from intel sgx. *IACR Cryptology ePrint Archive*, 2017:549, 2017.

[70] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, page 513–528, USA, 2010. IEEE Computer Society.

[71] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[72] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[73] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. Multise: Multi-path symbolic execution using value summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 842–853, New York, NY, USA, 2015. ACM.

[74] Fahad Shaon, Murat Kantarcioglu, Zhiqiang Lin, and Latifur Khan. Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1211–1228, New York, NY, USA, 2017. Association for Computing Machinery.

[75] Bhargava Shastry, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. Towards vulnerability discovery using staged program analysis. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 78–97, Cham, 2016. Springer International Publishing.

[76] Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *International Conference on Fundamental Approaches to Software Engineering*, pages 261–275. Springer, 2008.

[77] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 299–310, New York, NY, USA, 2013. ACM.

[78] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Proceedings of the 12th International Conference on Static Analysis*, SAS'05, pages 352–367, Berlin, Heidelberg, 2005. Springer-Verlag.

[79] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. Stealthdb: a scalable encrypted database with full sql query support. *Proceedings on Privacy Enhancing Technologies*, 2019(3):370–388, 2019.

[80] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, January 1996.

[81] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 235–252, Vancouver, BC, 2017. USENIX Association.

[82] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 215–226, New York, NY, USA, 2014. ACM.

[83] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

[84] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 631–647, New York, NY, USA, 2016. ACM.

[85] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 85–96, New York, NY, USA, 2012. ACM.

[86] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

[87] Danfeng Zhang and Daniel Kifer. Lightdp: Towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 888–901, New York, NY, USA, 2017. ACM.

[88] Kehuan Zhang, Zhou Li, Rui Wang, XiaoFeng Wang, and Shuo Chen. 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*, CCS '10, pages 595–606, New York, NY, USA, 2010. ACM.

[89] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, 2017. USENIX Association.