SPLICE: Efficiently Removing a User's Data from In-memory Application State

Xueyuan Han
vanbasm@wfu.edu
Wake Forest University
Winston-Salem, NC, USA

James Mickens
mickens@g.harvard.edu
Harvard University
Cambridge, MA, USA

Siddhartha Sen
sidsen@microsoft.com
Microsoft Research
New York, NY, USA

ABSTRACT

SPLICE is a new programming framework that allows security-conscious applications to efficiently locate and delete a user's in-memory state. The core technical challenge is determining how to delete a user's memory values without breaking application-specific semantic invariants involving the memory state of remaining users. SPLICE solves this problem using three techniques: taint tracking (which traces how a user's data flows through memory), deletion by synthesis (which overwrites each user-owned memory value in place, replacing it with a value that preserves the symbolic constraints of enclosing data structures), and a novel type system (which forces applications to employ defensive programming to avoid computing over synthesize-deleted values in unsafe ways). Using four realistic applications that we ported to SPLICE, we show that SPLICE's type system and defensive programming requirements are not onerous for developers. We also demonstrate that SPLICE's run-time overheads are similar to those of prior taint tracking systems, while enabling strong deletion semantics.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Data anonymization and sanitization*; Information flow control.

KEYWORDS

In-memory deletion, taint tracking, constraint solvers, data structures, defensive programming

ACM Reference Format:
Xueyuan Han, James Mickens, and Siddhartha Sen. 2023. SPLICE: Efficiently Removing a User's Data from In-memory Application State. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3576915.3623070

1 INTRODUCTION

In most online applications, servers track per-user state. This state often involves sensitive aspects of a user's activity, and often resides in server memory. For example, the server side of a social networking application manages a user's posts; the posts might be stored in full on disk, with portions being fetched into RAM on demand in response to user requests. As another example, the server side of a VPN application might have an in-memory routing table which maps a user's incoming network stream to an outgoing one that connects to a destination host. Applications also frequently use server-side memory to store cryptographic keys (e.g., belonging to TLS sessions).

This sensitive information is an attractive target for attackers who possess memory exploits like Heartbleed [6] or Cloudbleed [50]. Unfortunately, such exploits remain common in modern software, as evidenced by the many CVEs that continue to involve memory vulnerabilities (e.g., [43–46]). Memory disclosures are particularly troubling for security-conscious applications like VPNs [24, 49], private messaging platforms [2, 70], mix nets [19], and analytics tools for private social graphs [69], because these services intrinsically deal with sensitive data that users want to hide from prying eyes.

Unfortunately, finding and deleting a user's in-memory data is hard [72]. Standard programming languages and run-time frameworks do not provide ways to easily splice a particular user's data *and its derivatives* from arbitrary in-memory data structures. Simply terminating all server-side processes (thereby deleting all of their in-memory state) is typically unattractive for reasons of availability, consistency, and performance. For example, client connections might drop, and in-memory caches must be rewarmed with data belonging to non-deleted users. Fixing this collateral damage would require server activity that is often expensive in terms of computation, IO, and thus client-perceived response latency [39, 75]. Recovery code is also complex and may trigger recovery storms [29, 30] or additional faults [26]. Thus, intentional crashes are an unsatisfying mechanism for surgical deletion of in-memory state.

If servers only kept encrypted in-memory data, then that data could be effectively deleted by destroying the associated key [67]. However, even in privacy-focused applications that handle end-to-end encrypted user data, server RAM often contains unencrypted *metadata* about encrypted user state. For example, even though a Tor router [19] never sees cleartext client traffic, the router maintains next-hop routing metadata for all of the network streams that pass through it; the metadata includes sensitive information such as the IP addresses for a stream's endpoints. Being able to efficiently and completely remove such cleartext metadata is important for privacy-focused services in which a user may suffer real-world harm if server-side evidence of the user is found.

In this paper, we introduce SPLICE, a new programming framework that helps developers of security-conscious applications to track and delete the in-memory state belonging to particular users. SPLICE leverages three ideas: taint tracking, deletion via synthesis, and defensive programming.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623070

Taint tracking: After receiving data from a user, taint tracking [22, 73] allows Splice to observe how that data (and its derivatives) flow to various memory locations.

Deletion via synthesis: Later, when a user requests to be deleted from a service, Splice finds each value that is tainted by the user and deletes it. Naïvely deleting a value by overwriting it with a static tombstone (e.g., `null`) or a random value would break data structure invariants, rendering applications unusable. Keeping the value in memory but marking it as poisoned (to prevent the application from accidentally using it) would leave sensitive memory data vulnerable to leakage via careless programming or memory exploits such as Heartbleed. Many data structures naturally define methods to remove items; however, executing those methods might further propagate taint or shuffle values throughout memory, e.g., when nodes in a binary tree are rotated to restore balance after a deletion. This activity may force Splice's memory traversal to restart (lest the traversal miss tainted values that reside in locations that were previously untainted). To avoid these problems, Splice overwrites a value to delete, replacing the original data with synthesized data that satisfies the structural constraints of enclosing data structures. Splice creates the synthesized data by invoking a constraint solver; the constraints are derived from data-structure-specific invariants provided by the developer of the data structure. Splice pre-annotates the data structures provided by a language's standard library, and provides a DSL which programmers can use to annotate custom data structures.

Defensive programming: A synthesis-deleted value preserves data structure invariants, ensuring that, e.g., C++-style iterators still operate as expected. However, a synthesis-deleted value is "fake"—a program should not externalize it to clients or use it to decide how the program manages the state of undeleted users. To ensure these safety guarantees, Splice leverages the defensive programming model that is already popular in many web applications [3, 20, 51, 58]. These applications already treat user-submitted data as suspicious, requiring it to be sanitized before further use by the program. Splice extends this idea to data structures that are *internal* to the program. In particular, data structures that hold Splice-deletable information must be treated as suspicious. Splice enforces this policy using a type system which distinguishes between *trusted* and *untrusted* variables. Trusted variables are guaranteed to never hold synthesized data. An untrusted variable is associated with a `.synthesized` flag that indicates whether the variable contains synthesized data (and therefore corresponds to a synthesis-deleted value). Splice's type system ensures that (1) only trusted values are externalized via IO mechanisms, and (2) if the right-hand side of an assignment statement is untrusted, then the left-hand side will be untrusted as well unless the developer explicitly *sanitizes* the assignment. In this fashion, Splice can generate compile-time errors in statically-typed languages (or run-time errors in dynamically-typed languages) that prevent synthesized data from affecting trusted variables or trusted output sinks.

We have modified a Python runtime to implement Splice's type system and Splice's heap traversal algorithm for locating a particular user's data. We also built a standard library of common data structures that are pre-annotated with synthesis constraints. We ported four applications to run atop this Splice environment: a Django-based e-commerce app, a lightweight Twitter clone, a VPN service, and a privacy-oriented rendezvous server for peer-to-peer communication. Overall, we found the porting effort to be tractable. As we demonstrate in §6, Splice's computational overheads during normal operation are similar to those of other taint tracking systems, but enable a new benefit: the ability to enumerate and remove all of a user's in-memory state. Splice's synthesis-based deletion relies on constraint solvers, which can be slow; however, by analyzing the dependencies between the constraints of different variables to delete, Splice aggressively identifies opportunities to synthesize different variables in parallel, reducing synthesis-deletion latency by 21%–77%.

In summary, our contributions are as follows:

- a new, synthesis-based model for deleting specific pieces of sensitive in-memory data;
- an algorithm that employs taint tracking and heap traversals to tag in-memory data with the associated users and later find the data associated with a particular user;
- an implementation of these approaches in Python, and a set of ported applications which demonstrate that Splice's abstractions are usable and Splice's performance is reasonable for applications that want to provide strong semantics for the deletion of in-memory data. Splice incurs a 1.31×–2.69× increase in CPU utilization, and a 1.26×–1.37× increase in memory utilization, with deletion times ranging from a few milliseconds to a few tens of seconds for the most complicated applications.

Splice is complementary to work that provides deletion semantics for on-disk SQL data [37, 60, 64]. By combining Splice with one of those systems, developers get a full-stack solution for removing a user from the server-side of an application.

2 THREAT MODEL

A distributed system like a web service is composed of clients and servers. Splice focuses on the server-side deletion of user-derived, in-memory data. Splice is only concerned with post-deletion attackers who can examine server-side memory *after* deletion occurs; preventing information leakage to *pre*-deletion attackers is out of scope for Splice.

After Splice has deleted a user's state from the memory of server-side processes, remnants of that state may still persist on the user's client devices (e.g., in the memory of a web browser or smart phone app). The user can remove their client-side in-memory state by killing the relevant program (e.g., by closing a browser tab or terminating an app). We assume that the user would naturally perform such actions after submitting a Splice deletion request; the client-side portion of the web service can remind users to do so after receiving a deletion request. Deleting service-related disk state on the client-side is out of scope for Splice, but Splice is compatible with prior approaches for removing or hiding this state in ways that are stronger than just deleting an app's files [4, 14, 55]. On the server-side, Splice does propagate taint to disk when in-memory data is persisted; see Sections 3.2, 3.4.3, and 5.

Even after the user has removed their state from the server side and their client side, remnants may still exist in the memory or persistent storage of another client (e.g., because another user's web browser still holds an in-memory copy of a photo that the user wants to delete). This information leakage is out of scope for Splice.

However, from the privacy perspective, deleting the user's state on servers and on the user's client devices is typically much more important than deleting the user's state on another user's client devices. Removing the user's server-side data is urgent because, in a centrally-managed web service, datacenter machines store all of the user's information and all of the context that ties the information to other users and other aspects of the service (e.g., billing). Removing the user's state on their own client is urgent because, if the user is subject to harassment from a government or another party, that party will likely have a particular interest in examining the user's specific device. In contrast, the harassing party may not be able to easily determine which devices belonging to other users contain data associated with a specific individual.

On the server side, even if Splice overwrites application-visible memory copies of sensitive data, additional copies may exist in memory buffers that are not directly accessible to application-level code. For example, copies of sensitive strings may exist in unflushed `libc` buffers, or in outbound network packets that are queued for transmission. Deleting sensitive information in these buffers is out of scope for Splice, although Splice composes well with prior solutions for scrubbing such buffers [9].

Splice assumes that developers are honest but imperfect. These developers explicitly want to create applications that correctly delete a user's data upon request; consequently, they opt into building applications atop the Splice framework. However, because developers are imperfect, they might forget to use defensive programming to validate memory state corresponding to possibly-deleted information. Splice's taint tracking and type system prevent developers from allowing unvalidated information to corrupt applications. Our Splice prototype does not propagate taint via implicit flows [59], but is compatible with approaches that do so [35, 57].

3 DESIGN

In this section, we describe Splice's design in more detail. First, we explain the programming model that Splice presents to developers; in particular, we explain Splice's type system, and how it allows developers to write defensive code that properly handles synthesize-deleted values (§3.1). We then describe how Splice employs taint tracking to assign ownership to in-memory state (§3.2). Then, we discuss how Splice locates the necessary state to remove upon receiving a deletion request from a particular user (§3.3). Finally, we describe how Splice associates symbolic constraints with data structures, and how Splice uses those constraints to synthesize-delete the memory values to remove (§3.4).

Our Splice prototype is built atop the Python runtime, and throughout this section, we often make Splice's design more concrete by referring to how a particular Splice concept is implemented for the Python runtime. However, we also describe how Splice's approach would work in a statically-typed language. Regardless of whether a language is dynamically-typed or statically-typed, Splice does require the language to be strongly-typed; strong typing ensures the correctness and safety of Splice's heap traversals and value deletion via value overwriting.

3.1 Type System

Splice's type system is inspired by the notion of *defensive programming* [3, 20, 51, 58], a safety-oriented programming style that is already common in web applications. Defensive programming requires applications to treat all user-submitted data as suspicious until proven otherwise. For example, when a user submits text-based content, server-side code should *sanitize* that content, e.g., removing embedded HTML tags or SQL commands that would allow a malicious user to inject code into the server-side execution flow [31, 71]. Defensive programming has traditionally been applied only to information pulled from *outside* a process. However, Splice additionally requires developers to apply defensive programming to certain data structures that reside *within* a process. In particular, internal data structures that may hold user data must be accessed defensively, because Splice may have synthesize-deleted information within that data structure. Synthesized data should not be externalized via IO devices, or consulted to influence decisions involving the data of non-deleted users. As explained below, Splice's type system forces programs to act defensively; if programs do not, they will encounter errors at compile-time or run-time (depending on the language).

Primitives: A programming language defines primitive types (e.g., integers, floating point numbers, and individual characters) as well as aggregation types (e.g., structs, objects, strings, and other types that bind a collection of primitives and/or aggregation types). For each primitive type, Splice differentiates between two derivative types: a *trusted* variant and an *untrusted* variant. A variable having the trusted variant will never contain synthesized data. A variable having the untrusted variant *may* contain a synthesized value, and therefore must be handled with caution, because the value is a fake one that Splice generated to satisfy a data structure invariant (§3.4). Splice allows a program to

- read an untrusted variable's value;
- check whether an untrusted variable actually contains synthesized data; and
- use the value of an untrusted variable in the right-hand side of an assignment to a different untrusted variable.

However, Splice prevents the direct assignment of an untrusted value to a trusted variable. Instead, such an assignment must be mediated by a *sanitization* operation which converts the possibly-synthesized value to one that has a trusted type. Only trusted types are safe to externalize outside the process via IO; trusted types are also safe for a program to consult when making decisions that would impact non-deleted users. Splice allows branching on an untrusted value, or assigning an untrusted value to an untrusted variable, because Splice's notion of safety is defined purely with respect to output externalization and assignments to trusted variables.

In a statically-typed language like Go, Splice introduces an untrusted variant of each built-in primitive type. For example, in Go, Splice would add an untrusted `int32_ut` type to complement the preexisting `int32` type; Splice would treat `int32` as the trusted variant of the 32-bit integer type. Using this approach, attempts to assign untrusted values to trusted variables or IO sinks are detectable at compile-time, just like any other type conflict. To enable sanitization functions (i.e., functions that derive trusted values from untrusted inputs), Splice uses a language's standard type conversion operators. For example, given an untrusted Go variable `u`, the code `var t int32 = int32(u)` enables the bytes in `u` to be assigned to a trusted variable `t`. Similar to how the standard version of Go disallows implicit type conversions, Splice also prohibits implicit

conversions from untrusted to trusted values. This policy forces developers to explicitly invoke sanitization code when updating trusted variables with state derived from untrusted values. Static checks also prohibit untrusted data from exiting the program via IO sinks like files, because those sinks only accept trusted values as inputs.

In general, programs should only employ untrusted-to-trusted casts when sanitizing untrusted values that are about to be externalized via IO sinks. Programs should otherwise not abuse untrusted-to-trusted casting to shove user data into trusted variables. The reason is that Splice's programming model assumes that potentially-deletable user data only exists within instances of untrusted types.

In a dynamically-typed language like Python, Splice also defines untrusted analogues of the trusted built-in primitive types. Similarly, Splice provides language-level mechanisms to cast an untrusted value to a trusted one. However, the Splice-modified runtime (not the compiler) is responsible for ensuring that assignment statements involving untrusted right-hand sides properly result in untrusted left-hand sides.

In both statically-typed languages and dynamically-typed languages, IO sources (e.g., network sockets) that may return user data provide an initial source of untrusted values. After fetching those values, the application computes over them, possibly deriving new untrusted values.

In most cases, Splice does not have to pay special attention to structs and other object types which aggregate primitive values. Reads and writes of individual fields in these objects are governed by the same type rules that we described above. However, as we explain soon, aggregation types (and individual primitives) that represent *OS abstractions* like processes and network sockets must be handled specially.

OS abstractions: Different languages use different interfaces to represent OS abstractions likes processes, sockets, and disk files. However, modern languages tend to use a specific object type to encapsulate information about an instance of a particular OS abstraction. For example, in Python, a new process is launched by creating a `Popen` object. That object defines fields like `.args` (for the process's command line arguments), `.pid` (for its process id), and `.returncode` (to store the exit value generated when the process dies). Go's `Process` and `ProcessState` structs aggregate similar information.

Objects corresponding to OS abstractions require special care at deletion time. For example, consider a process `P` which belongs to user Alice. When Alice requests that she be removed from a service, `P` must be removed. However, simply destroying `P`'s address space is insufficient, because the remnants of `P` in the parent process must also be removed. So, if the application is written in Python, Splice must synthesize-delete the relevant fields in the `Popen` object that represents `P` in the parent process, overwriting `.pid` and `.args` and so on with synthesized data.

Splice provides drop-in replacements for object types like `Popen`. Each replacement type is associated with *deletion manager* code. This code, implemented by Splice, performs the necessary clean-up activity when Splice determines that the associated OS abstraction must be deleted. For example, in Python, the deletion manager for a `Popen` process synthesize-deletes fields like `.pid`. Section 3.2 describes how Splice associates an instance of a system abstraction with a particular user.

Data structures: A data structure is just a collection of aggregation instances that are connected via references to each other. In a statically-typed language, data structures for storing user data will hold untrusted types; furthermore, data structure methods which retrieve information from those data structures will return untrusted types. In this way, the compiler can detect when a program is not being vigilant about handling a user-derived value that may have been synthesized.

In a dynamically-typed language like Python, IO sources return untrusted data, just like in a statically-type language. However, the runtime (not the compiler) ensures that assignments involving untrusted right-hand sides will generate untrusted left-hand sides. Enforcement of this rule is sufficient to guarantee that, when user-derived values enter a data structure, those values will be untrusted. The runtime is also responsible for preventing untrusted values from being externalized via IO sinks; for example, Splice's Python runtime throws an `UntrustedExternalizationException`.

In a data structure, aggregation types are connected via reference types. For example, in a unidirectional linked list, a `ListNode` aggregation type contains a `ListNode.next` reference to the following node in the list. In Splice, references like `.next` always use trusted types, and thus will never be synthesize-deleted. In other words, Splice does not define untrusted variants of reference types. Splice uses this approach because changing the shape of a data structure at deletion time might propagate taint in undesirable ways (§1). Keeping the shape of data structures constant at deletion time also reduces the number of constraints that Splice must consider when synthesize-deleting values.

3.2 Taint Tracking

Taint tracking allows Splice to observe how a user's data propagates throughout memory. At the conceptual level, a taint tracking system consists of three parts: a source of initially tainted data, rules for propagating taint to individual program variables, and higher-level taint policies which define how tainted data is permitted to flow to various sinks like processes and sockets.

Sources of initially tainted data: Splice primarily targets network servers. Thus, the major sources of tainted data are network sockets which communicate with end-user clients. Splice relies on application-specific mechanisms to taint incoming socket data with the appropriate user id. For example, in a routing indirection service like Tor, a user id might correspond to a client IP address. In an email application, a user id might correspond to an email address. Splice provides drop-in replacements for language-level socket read interfaces like `socket.recv()` in Python; these replacement interfaces add a parameter that specifies the user id which Splice should assign as the taint of the returned bytes.

Taint propagation rules: Splice uses dynamic taint tracking [12]. At a high level, each memory address that is visible to the programmer is assigned a taint tag; tags are stored in memory that is only accessible by the runtime. As a program executes, the taint tracking framework interposes on assignment statements, such that the left-hand side of an assignment receives the unions of the taints of the values on right-hand side. For example, the assignment `lhs = rhs1 + rhs2` results in `lhs` receiving the union of `rhs1`'s taint and `rhs2`'s taint. Our prototype uses taint propagation rules that

are similar to those of TaintDroid [22]. The curious reader can see our rules in Table 1.

Higher-level taint policies: Splice allows tainted data to flow to client devices since, according to our threat model (§2), Splice is not opinionated about client-side information flows. Splice allows tainted data to flow to server-side persistent storage; as is standard in taint tracking systems, Splice assumes a file system that can associate files with taint tags (e.g., at the granularity of whole files [22] or byte ranges [76]). Using such a file system, Splice ensures that files that are written using tainted in-memory data receive the associated taint. Splice also ensures that, when those files are read back into memory, taint propagates to memory values as expected.

Splice also needs to assign taint to *execution contexts* like processes and threads. Splice assumes that each thread is associated with (i.e., tainted by) at most one user, although a thread may compute over data belonging to multiple users, and propagate taint as described above. Splice assumes that each process is also associated with at most one user. A thread or process is associated with no user if the thread acts on behalf of the service itself. This system model supports application models in which a server spawns initial processes and threads at launch time, and then creates new, per-user threads and/or processes in response to an incoming request for a particular user.

Splice requires developer assistance to identify whether a newly-spawned execution context should be associated with a particular user. Splice provides drop-in replacements for language-level interfaces for creating execution contexts; the Splice versions of those interfaces provide additional parameters that allow developers to specify the user id (if any) for the execution context. For example, in our Python-based Splice prototype, we extend the interfaces for creating a process, `subprocess.Popen`, to accept taint identifiers. A deletion request sent to any process in a process tree will induce Splice to delete a user's state from all of the address spaces belonging to the process tree.

3.3 Finding Values to Delete

As a Splice-enabled process executes, memory locations receive tainted user values. When a particular user asks to be removed from the process, Splice must find all of the memory values that are tainted by the user. At a high level, Splice does so by traversing memory in the same style as a garbage collector [34]. Starting from a set of heap roots, Splice recursively follows memory references, exploring the graph of memory values that are reachable from each root. If a discovered value has the taint of the user to remove, Splice must decide how to delete the value. If the value is an object that provides a deletion manager (§3.1), Splice invokes the manager to trigger type-specific deletion code. Otherwise, for primitive values or aggregation-of-primitives objects, Splice consults each value's synthesis constraints to determine how to overwrite the value (§3.4).

Note that a memory value may be tainted by multiple users. For example, consider the assignment `lhs = rhs1 + rhs2`, where `rhs1` is tainted by Alice and `rhs2` is tainted by Bob. The resulting value that is stored in `lhs` will be tainted by both Alice and Bob. Later, if Bob requests to be deleted, Splice's memory traversal will eventually find the `lhs` value. Splice will see that the value is multi-tainted, and will not synthesize-delete the value. Instead, Splice will simply remove Bob's taint from the value. This policy has the practical advantage that, when Bob is removed from the service, the collateral damage to the rest of the application is minimized—only state that solely belongs to Bob is removed. However, this policy does allow some information leakage (§3.5).

Deletion by synthesis overwrites memory locations with new values. Thus, at deletion time, Splice must be careful to avoid race conditions with application threads that want to access soon-to-be-deleted state. To avoid race conditions, Splice only performs synthesis-deletion during *quiescent periods* when other application threads are inactive. Our Splice prototype provides two concrete definitions for quiescence, each of which is suited for a different way of organizing computation inside a process.

- For a single-threaded, event-driven process (e.g., that employs `libevent` [40] to handle asynchronous IO), Splice's deletion thread can safely run between the dispatch of event handlers.
- For a multithreaded process, Splice adds a global reader/writer lock. An application thread acquires the lock in reader mode in the top-most function in its call chain; the acquisition should occur before the thread tries to access any local variables, heap variables, or global variables. The application thread releases the lock when the thread's call chain finishes, or when that chain returns to the beginning of a loop in the top-most function. Splice's deletion thread acquires the lock in writer mode upon receiving a user request for deletion. The deletion thread releases the lock once the heap has been traversed and the necessary memory values have been synthesize-deleted.

Both approaches enforce the same policy: at any given moment, either Splice's deletion thread is running, or zero or more application threads are running. Thus, Splice will never synthesize-delete a value that is actively being used by an application thread. We are investigating techniques from concurrent garbage collection [10, 56] that may allow Splice's deletion thread to safely run in parallel with application threads.

Recall that Splice requires applications to be written in strongly-typed languages. This requirement ensures that, when Splice performs a deletion-time memory traversal, Splice can determine with perfect accuracy the type of a memory value, whether the value belongs to the user to delete, and what its associated metadata is (in particular, the value's synthesis constraints (§3.4) and deletion manager (§3.1), if any).

3.4 Deletion via Synthesis

Splice must delete a particular user's values in place while preserving the semantic integrity of the state belonging to other users. Doing so in a fully automated way is hard, since applications can use arbitrary data structures to implement arbitrary application-level semantics. Thus, Splice requires assistance from developers. In particular, Splice requires developers to annotate their data structures with *symbolic constraints*. These constraints explain how each value in the data structure[1] relates to other values in the data structure. When Splice must delete a particular value, Splice concretizes the value's symbolic constraints, and then asks a constraint solver to produce a new value that satisfies those constraints. Splice overwrites the old value with the synthesized one, thereby deleting the user's old state.

[1] Besides references–see §3.1.

Op Type	Op Semantics	Taint Propagation	Description
Unary Immutable	$o_A \leftarrow$*self*.op()	$\mathcal{T}(o_A) \leftarrow \mathcal{T}(self)$	Set o_A's taint to *self*'s taint
Binary Immutable	$o_A \leftarrow$*self*.op(o_B)	$\mathcal{T}(o_A) \leftarrow \mathcal{T}(self) \cup \mathcal{T}(o_B)$	Set o_A's taint to union of *self*'s taint and o_B's taint
Binary Mutable	*self* $\leftarrow$*self*.op(o_A)	$\mathcal{T}(self) \leftarrow \mathcal{T}(self)\mathcal{T}(o_A)$	Update *self*'s taint with o_A's taint
N-ary Immutable	$o_A \leftarrow$*self*.op($o_B, \ldots, o_N$)	$\mathcal{T}(o_A) \leftarrow \bigcup(\mathcal{T}(self), \mathcal{T}(o_B), \ldots, \mathcal{T}(o_N))$	Set o_A's taint to the union of *self*'s, o_B's, ..., and o_N's taint
N-ary Mutable	*self* $\leftarrow$*self*.op($o_A, \ldots, o_N$)	$\mathcal{T}(self) \leftarrow \bigcup(\mathcal{T}(o_A), \ldots, \mathcal{T}(o_N))$	Update *self*'s taint with the union of o_A's, ..., and o_N's taint
Class Immutable	$o_A \leftarrow C$.op($o_B, \ldots, o_N$)	$\mathcal{T}(o_A) \leftarrow \bigcup(\mathcal{T}(o_B), \ldots, \mathcal{T}(o_N))$	Set o_A's taint to the union of o_B's, ..., and o_N's taint
Iteration	*for* o_A in o_B	$\mathcal{T}(o_A) \leftarrow \mathcal{T}(o_B)$	Set o_A's taint to o_B's taint
Collection Input	$o_A \leftarrow$*self*.op($o_B, \ldots, <o_N, \ldots, o_Z>$)	$\mathcal{T}(o_A) \leftarrow \bigcup(\mathcal{T}(self), \mathcal{T}(o_B), \ldots, \mathcal{T}(o_N), \ldots, \mathcal{T}(o_Z))$	Set o_A's taint to union of *self*'s, o_B's, ..., o_N's, ..., and o_Z's taint.
Collection Return	$<o_A, \ldots, o_M> \leftarrow$*self*.op($o_N, \ldots, o_Z$)	$\mathcal{T}(o_A) \leftarrow \bigcup(\mathcal{T}(self), \mathcal{T}(o_N), \ldots, \mathcal{T}(o_Z)), \ldots, \mathcal{T}(o_M) \leftarrow \bigcup(\mathcal{T}(self), \mathcal{T}(o_N), \ldots, \mathcal{T}(o_Z))$	Set o_A's, ..., o_M's taint to union of *self*'s, o_N's, ..., and o_Z's taint.

Table 1: Splice's taint propagation rules. In the lefthand column, a "collection" can be `list`, `tuple`, or `dict`.

3.4.1 Overview. In Splice, each program variable (regardless of whether it is trusted or untrusted) is associated with zero or more taints; each taint represents a user whose data influenced that variable's value. An untrusted variable is also associated with a symbolic constraint which explains how to overwrite that value if the value later requires deletion-by-synthesis.

Splice differentiates between two kinds of assignments to a primitive: *constraining assignments* and *maintaining assignments*. Constraining assignments add more constraints to a variable (i.e., a memory location). Maintaining assignments do not. Constraining assignments occur in two scenarios:

- Constraining assignment happens when a value is first initialized using the default constraints for the value's type. For example, the local variable assignment `uint32_ut lhs = 50` results in `lhs` receiving the constraints `gt(0) AND lt(`2^{32} `- 1)`, meaning that the variable could be overwritten by any valid `uint32` value. The variable receives the loosest possible constraints for the `uint32` type because the variable is not associated with any data structure whose invariants depend on `lhs` (and thus restrict the possible values that `lhs` could be overwritten with).
- As hinted above, constraining assignment occurs when a value is inserted into a data structure such that, post-insertion, the data structure invariants depend on the value. For example, suppose that after the assignment `uint32_ut lhs = 50`, a program does `binarySearchTree.insert(lhs, anotherLhs)`, with `lhs` used as the insertion key, and `anotherLhs` being the data which the tree associates with the insertion key. If `lhs` is passed by reference to `.insert()`, then the single in-memory copy of `lhs` has its constraints updated; otherwise, the new pass-by-value copy of `lhs` has its constraints updated. In either case, the new high-level constraints become "any valid `uint32` value, where that value must be greater than any value in `lhs`'s left subtree, and less than any value in `rhs`'s right subtree." The new "where that . . ." text represents the constraining aspect of the assignment. The new constraints are generated by Splice, who inspects the symbolic annotations belonging to the binary search tree and applies the appropriate new constraints to `lhs`. Note that `anotherLhs` (i.e., the data which the tree associated with the insertion key) does *not* receive a constraining assignment when it is added to the tree node by value or by reference. The reason is that the tree's invariants do not depend on the value of `anotherLhs`.

Assigning one struct to another using copy-by-value results in a maintaining assignment of each right-hand struct field to the associated field in the left-hand struct.

⟨*symb-const*⟩	::=	{ ⟨*cond-dnf*⟩ | ⟨*dnf*⟩ }-
⟨*cond-dnf*⟩	::=	'if' ⟨*cond*⟩ 'then' ⟨*dnf*⟩ { 'elif' ⟨*cond*⟩ 'then' ⟨*dnf*⟩ } ['else' ⟨*dnf*⟩]
⟨*dnf*⟩	::=	⟨*cnf*⟩ { 'OR' ⟨*cnf*⟩ }
⟨*cnf*⟩	::=	⟨*logic-const*⟩ { 'AND' ⟨*logic-const*⟩ }
⟨*logic-const*⟩	::=	('gt' | 'ge' | 'lt' | 'le' | 'eq' | 'ne') '(' { ⟨*cond*⟩ }- ')'
⟨*cond*⟩	::=	cond_op '(' { ⟨*cond*⟩ } ')'
	|	variable

Table 2: DSL for defining symbolic synthesis constraints.

Perhaps counterintuitively, when Splice encounters a primitive assignment like `uint32_ut lhs = rhs1 + rhs2`, Splice does not somehow propagate the combination of `rhs1`'s constraints and `rhs2`'s constraints. Instead, `lhs` receives the loosest possible constraints for the `uint32_ut` type. The reason is that, until `lhs` becomes associated with a data structure and entangled with its structural invariants, Splice can synthesize-delete it by overwriting it with any valid `uint32` value; no data structure invariants could possibly be broken by such an assignment, and defensive programming (§3.1) will prevent the overwritten (and fake) value from being used in a meaningful way by the program.

Given that high-level overview, we now provide specific examples of how data structures should be annotated with symbolic information (§3.4.2 and §3.4.3). We then explain how to synthesize-delete multiple values in parallel without breaking application invariants (§3.4.4).

3.4.2 Classic Data Structures. Many programming languages provide libraries that define commonly-used data structures like dictionaries and search trees. These data structures typically have simple constraints. Furthermore, the constraints can be provided by the language designers, freeing application developers from the responsibility of generating them.

Example 1: Binary search tree. Consider a node in a binary search tree which stores unique keys. The constraint on the node's sorting value is that the value must be greater than any value in the left subtree, and less than any value in the right subtree. Splice uses a simple domain-specific language (Table 2) to express the constraint as follows:

$$gt(predecessor(self)) \textbf{ AND } lt(successor(self))$$

where `self` refers to the sorting value, `gt()` means "greater than," and `lt()` means "less than." `predecessor()` and `successor()` are predicates, written by the developer of the data structure, that

return the values in the tree that are immediately smaller than and greater than a given value in the tree. Predicates are defined in the application language; for example, a Python tree expresses predicates using Python code. At run time, when Splice needs to synthesize-delete a value, Splice finds the associated constraints, runs the predicates (allowing the enclosing constraints to be concretized), and then passes the concretized constraints (e.g., `gt(7) AND lt(29)`) to the constraint solver. The returned value of the constraint solver is used to overwrite the value to delete. For the corner case in which a predicate returns `nil` (e.g., because the value to delete is the largest value in the tree and thus has no `successor(self)`), the predicate is dropped from the set of constraints that Splice passes to the constraint solver.

The example above used a textbook binary search tree. However, the same constraints apply to more sophisticated kinds of sorted binary trees like red-black trees and AVL trees. These more complicated data structures have more complicated insertion and deletion code to ensure balancing invariants. However, Splice deletes values by overwriting them in place, meaning that, post-deletion, the shape of a tree is unchanged.

Example 2: Hash table. To synthesize-delete a key `k` in a hash table, Splice must be able to find a new `k'` with the same hash. Thus, insertion into a hash table results in a constraining assignment which adds this condition:

$$eq(hash(self'), hash(self)))$$

where `hash()` is the hash function used by the hash table. While Splice does not categorically prevent applications from using cryptographically strong hash functions, their use inside data structure code would pose challenges to the constraint solver at deletion time, since cryptographically strong hash functions are difficult to invert by design. Therefore, a Splice-amenable hash table must use a hash function that is tractable for the constraint solver to invert [65].

3.4.3 *More Complex Data Structures.*

Example 3: Redis sorted set. Redis [5] is a popular in-memory data store that provides a variety of classic data structures like lists and hash tables. Redis also provides more advanced structures like sorted sets. A sorted set contains two-tuples, where the first tuple element is a floating point number called the "score," and the second element is a string. Each tuple resides in a sorted list (used for iterating through the tuples in order of their scores) and a hash table (used to efficiently find the score for a particular key).

With Splice, insertion into a sorted set creates a new tuple, and generates three constraining assignments. The first constraining assignment involves the string that is used as a hash table key; the assignment is equivalent to the one shown in **Example 2**. The second and third constraining assignments involve the sorted list. The second restricts the possible values for the new tuple's score:

$$\begin{aligned}&\textbf{if } gt(prev(self), self) \textbf{ then } gt(prev(self))\\ &\qquad\qquad\qquad\textbf{else } ge(prev(self)) \textbf{ AND}\\ &\textbf{if } lt(next(self), self) \textbf{ then } lt(next(self))\\ &\qquad\qquad\qquad\textbf{else } le(next(self))\end{aligned}$$

The third constraining assignment restricts the possible values for the new tuple's string:

$$\begin{aligned}&\textbf{if } eq(prev(self), self) \textbf{ then } gt(prev(self)) \textbf{ AND}\\ &\textbf{if } eq(next(self), self) \textbf{ then } lt(next(self))\end{aligned}$$

The overall result is that the string receives two constraining assignments, and the score receives one. Note the use of the **if-else** construct, which allows Splice to choose at deletion time which constraints to pass to the constraint solver.

Example 4: Log-structured merge tree. Log-structured merge (LSM) trees [38] are used by persistent key/value stores like LevelDB [16] and RocksDB [63]. In this section, we describe the symbolic annotations for a LevelDB-style LSM tree.

Such a tree consists of an in-memory structure called a *memtable*, and a disk-based structure made of several levels of *sstables* ("sorted string tables").

- The memtable is a skiplist. The LSM tree uses it as a write log to hold recently updated key/value pairs. The skiplist also memorializes the removals of key/value pairs, logging each removal using a special tombstone tuple for the associated key.
- The sorted tables store colder key/value pairs. When the skiplist reaches a cutoff size (e.g., 4 MB in LevelDB), the skiplist is converted into an sstable. An sstable holds entries sorted by their keys, where each entry corresponds to a key/value pair or a key tombstone. Level 0 consists of the most recently generated sstables; these tables may have overlapping key ranges. However, there are no overlaps at higher levels. For example, at Level 1, each sstable covers a unique key range. Periodically, a Level i table is removed from Level i and merged with Level $i+1$ by removing the overlapping Level $i+1$ tables, and creating new Level $i+1$ tables containing no duplicate keys. Thus, recent updates are gradually pushed down the levels, allowing old (i.e., merged) sstables to be garbage collected.

A read for key k is handled by scanning the memtable, then the Level 0 sstables, then the relevant Level i sstable, and so on until a match is found or no more sstables remain to be scanned.

Logically speaking, Splice treats the on-disk component of the LSM tree in the same way that it treats the in-memory part. When memtables are converted to sstables and spilled to disk, Splice serializes the associated Splice types, propagating taint and symbolic constraints. When sstables are pulled into memory during merging or to satisfy a read query, Splice deserializes the associated Splice types to ensure that taint propagates as expected and symbolic constraints are updated as needed.

The constraints for the memtable skiplist are straightforward:

$$gt(prev(self)) \textbf{ AND } lt(next(self))$$

The constraints for the sstables are more complicated, because keys in sstable entries are delta-encoded to save space. An sstable stores a full (i.e., non-delta-encoded) key at predetermined intervals; in between, each key $\mathtt{k}_i$ is represented as the length of the prefix shared with the previous $\mathtt{k}_{i-1}$, and the suffix string that is unique to $\mathtt{k}_i$. For example, suppose that Alice inserts the key "`try`", Bob inserts the key "`trying`", and then the memtable is immediately flushed to an sstable. The first sstable entry will contain the full key "`try`", and the second entry will contain the delta-encoded key `<3, "ing">`. As expected, the full key "`try`" will be tainted

by Alice, and the delta-encoded key <3, "ing"> will be tainted by Bob. Splice assigns the following symbolic constraints to a key:

$$gt(prev\_decoded\_key(self))\ \textbf{AND}$$
$$lt(next\_decoded\_key(self))\ \textbf{AND}$$
$$eq(prefix(get\_prefix\_len(next(self)), self),$$
$$\quad prefix(get\_prefix\_len(next(self)), self'))$$

The first two constraints ensure that the synthesized key k' maintains the total order of decoded keys in the sorted table. The third constraint ensures that k' preserves enough of the initial characters in k to not break the delta-encoding of the key following k.

Suppose that later, Alice issues a LevelDB operation to remove the key "try", causing a tombstone record to be placed in the memtable. During the subsequent merging of the memtable with sstables, LevelDB will realize that the key "try" has been removed, but is a prefix for the surviving key "trying". LevelDB will thus create a new, non-delta-encoded sstable entry for "trying", with the first three characters tainted by Alice, and the trailing three characters tainted by Bob. From the perspective of Splice's memory traversal (§3.3), a string possesses the union of the taints of its characters. If Alice requests a Splice-level deletion, Splice will find the multi-tainted string and remove Alice's taint from the relevant characters; however, Splice will not synthesize-delete the key. The synthesize-deletion is only triggered once Bob requests a Splice-level deletion.

3.4.4 Synthesizing in Parallel. Constraint solvers can be slow. To hide some of this cost at deletion time, Splice aggressively synthesizes multiple values in parallel. Define the *dependence set* of a value-to-delete v_x as the transitive closure of the values mentioned by v_x's constraints, or recursively mentioned by the constraints of those values. Splice can synthesize-delete v_x before, after, or in parallel with v_y if v_y is not in v_x's dependence set. However, if v_y is in v_x's dependence set, then v_y must be synthesize-deleted before v_x. Splice can synthesize v_x and v_y in parallel if neither resides in the dependence set of the other.

At deletion time, Splice traverses memory to find the necessary values to delete (§3.3). After finding all of those values, Splice builds dependency graphs for the values to delete, where the directed edge $v_y \rightarrow v_x$ means that v_y must be synthesized before v_x. After walking through memory, Splice has produced a collection of one or more graphs, where individual graphs are disconnected from each other. For two values residing in different graphs, Splice can synthesize-delete the values in parallel. Within the same graph, Splice must handle ancestors before handling descendants, but there may still be opportunities for parallelism. For example, if a graph is a tree, Splice could proceed level by level, synthesize-deleting in parallel all N nodes of a level (or C nodes if Splice only has access to $C < N$ CPUs). More generally, for a graph with multiple roots (i.e., multiple nodes with no incoming edges), Splice starts its parallel traversal from those roots, similar to how a topological sort would work [36].

In theory, a dependency graph can have cycles of arbitrary length. In practice, for the data structures that we examined, cycles were rare. The cycles that we did observe were always of length two, and the constraints associated with the two edges were mirrored versions of each other. For example, a binary search tree might have a dependency cycle of length two involving a parent and a left child of that parent. The parent-to-left-child dependency would be $v_p > v_c$, and the left-child-to-parent dependency would be $v_c < v_p$. For length-two chains of mirrored dependencies, Splice can break the chain arbitrarily (e.g., by removing the parent-to-left-child dependency) and synthesize using the resulting dependency graph without fear of breaking correctness.

3.5 Discussion

A single memory value can be multi-tainted, having the taints of (say) both Alice and Bob (§3.2). If Bob asks to be removed from the service, and Splice finds such a multi-tainted value, Splice will just remove Bob's taint, without synthesize-deleting the value. Doing so minimizes the collateral damage of deletion, but leaves more traces of Bob's former presence in memory. Splice could inform Bob of such undeleted data at deletion time. In theory, Splice could instead try to speculate about what the multi-tainted value *would have been* had Bob's inputs never been processed by the system; Splice would then overwrite the value with the result of the speculation. However, devising a rigorous way to imagine such counterfactual worlds is hard, so we leave a more thorough investigation of this idea to future work.

When a value must be synthesize-deleted, the value's constraints may be such that the value can only be overwritten by its current value. For example, suppose that a binary search tree of integers does not allow duplicates, and contains the values 6, 7, and 8. In this example, the value 7 can only be overwritten with 7, meaning that synthesis-deletion would not actually delete information. As in the multi-taint example from the previous paragraph, Splice could warn a user that some data was unable to be deleted because of overly-restrictive constraints.

The fact that Splice deletion does not change the shape of a data structure may leak information about the possible values that could have existed pre-deletion [47]. Accurately quantifying such leakages is difficult. One strategy might be to use entropy metrics to capture Splice's ability to synthesize-delete values that are "very different" from the original ones. We leave this information theoretic analysis to future work.

4 IMPLEMENTATION

Our Splice prototype extends the Python runtime, providing four new components: a type system, a synthesis module, a deletion module, and a standard library. Our *type system*, containing roughly 1,650 lines of code, uses Python metaprogramming to introduce drop-in replacements for built-in primitive types and strings. Each instance contains hidden `.synthesized` and `.taint` flags, and uses reflection to interpose on assignments and update flags as appropriate. Our new type system also contains drop-in replacements for objects corresponding to OS abstractions (§3.1).

Our *deletion module* runs when a user asks to be removed from a service. This module, containing roughly 200 lines of code, uses the `guppy3` memory analysis tool [48] to traverse the application heap and locate the objects to delete; our prototype currently uses a single thread to walk the heap. The deletion module invokes our *synthesis module* to generate the values needed to overwrite a user's

state. The synthesis module contains roughly 1,400 lines of code; it runs Splice's dependency analysis (§3.4.4) and invokes the Z3 constraint solver [15] to generate new values, synthesizing multiple ones in parallel when dependencies allow.

Our final component is a *standard library* of common data structures, pre-annotated with symbolic constraints. The library contains all of the data structures mentioned in §3.4, and several others like a clone of the Redis key/value store [5]. The library contains roughly 3,900 lines of code.

5 PORTED APPLICATIONS

We ported four different Python applications to Splice: a Django-based ecommerce site, a lightweight Twitter clone, a VPN server, and a signaling server for end-to-end encrypted chats. Overall, we found that the porting effort was quite tractable. For example, we were able to port the SaltyRPC codebase, with which we had no previous familiarity, in less than two person-hours; we added roughly 500 lines of code to introduce taint sources and sinks, and modified the application to use Splice-provided object types for OS abstractions like sockets.

Django: Django [25] is a popular server-side web framework. When a client-generated HTTP request arrives, the request traverses several layers of middleware before reaching the core application logic. The most relevant layers are the session middleware, which is early in the request ingress path, and the authentication middleware, which is later. The session middleware inspects a request's cookies, generating a new client session if necessary. However, a request is not associated with a user until the request hits the authentication layer; to identify the correct user, this layer consults the request's cookies (to map an already-logged-in user to the request) or the database (to initially log in a user). Each middleware layer is stateless, retaining no in-memory data about a request after the request has been handed to the next layer. After traversing the middleware stack, the request hits the core application logic. That logic uses a SQL database as a backend, with each table having an associated Python-level class. When Django reads a table row into memory, Django represents it using an instance of the associated Python class; similarly, Django updates a table by serializing the appropriate Python-level objects and writing them to disk. Django has a pluggable in-memory caching layer, with developers choosing from several implementations like memcached or Redis. Once the application logic has fetched data from the cache or the backend, the logic assembles the HTML response using a templating engine. The response then goes up another set of stateless middleware components before returning to the client.

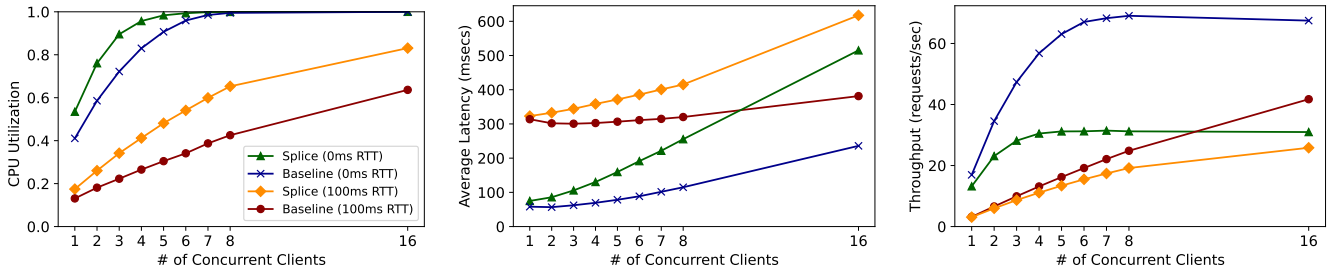
In the Splice version of Django, sockets generate untrusted, tainted data, with the taint being assigned by new code residing in the authentication middleware. Since all prior middleware in the ingress stack is stateless, not tainting a request earlier does not result in a loss of visibility into user-specific data. To propagate taint into the persistent storage layer, Splice transparently rewrites database schemas, adding additional columns to indicate whether a database value is synthesized; Splice considers all database values to be untrusted, so Splice does not add extra columns to indicate this. Splice rewrites the Python-level classes that reflect database rows, and also rewrites the database queries issued by Django to make them aware of the new schema. The rewritten queries use the data lineage approach of Cui et al. [13] to ensure that taint propagates correctly through SQL statements. When a Splice-level deletion of a database value is necessary, Splice nulls the associated value, unless the value is a primary key or used by a database index, in which case Splice deletes the whole row. Row deletion may trigger additional row deletions due to `FOREIGN KEY+CASCADE` relationships. We wrote a Python-based Redis clone to act as Django's in-memory caching layer.

In the request ingress stack, we added new middleware to handle Splice deletion requests. Django is multithreaded, so our deletion middleware uses the reader/writer approach from §3.3. In the response egress stack, we added new middleware which ensures that untrusted data is not externalized to clients.

MiniTwit: MiniTwit [53] is a Twitter clone that runs atop the Flask web framework [52]. MiniTwit supports key Twitter functionality like posting messages and following users. The original MiniTwit code employed a SQLite database to store profile information for registered users (e.g., email addresses and passwords). In our Splice port, we swapped out SQLite for our own Python implementation of LevelDB (§3.4.3) to demonstrate Splice's ability to taint and delete on-disk values that are not managed by a relational database. We also reused our Django ORM framework to allow MiniTwit to interact with the PostgreSQL database which stores tweets. Flask uses a multithreaded, middleware-based design that is conceptually similar to that of Django; so, we could apply our porting strategy for Django to Flask in a straightforward way.

SSTP: SSTP (Secure Socket Tunneling Protocol) is a VPN protocol for encapsulating Layer 2 traffic within a TLS connection [41]. We ported an open-source SSTP server [7] to Splice. Unlike Django and MiniTwit, where a taint tag corresponds to an application-level username, the SSTP server associates each taint tag with a client IP address. After a client establishes a connection with the SSTP server's parent process, the server forks a child process which immediately `exec()`s an instance of `pppd` [1]. An SSTP client forwards an outbound Layer 2 packet via the TLS tunnel to the appropriate `pppd` child process; in turn, the child forwards the packet to the parent process via a pipe, and the parent sends the rewritten packet to the destination server. Packets received from the remote endpoint follow the reverse path.

SaltyRTC: SaltyRTC is a protocol for establishing peer-to-peer, end-to-end encrypted communication [28]. We ported the SaltyRTC signaling server, which acts as a rendezvous point for *initiators* (who define "connection paths" at the server) and *responders* (who connect to paths to exchange information with the associated initiators). A path is just a hex representation of the initiator's public key. Once two peers have authenticated to the server and linked themselves to the same path, the peers exchange messages directly (i.e., without forwarding assistance from the server). The server tracks connection state via an in-memory dictionary which uses initiator public keys as lookup keys; each lookup key is associated with connection metadata for one or more responders who are talking to the initiator. Similar to SSTP, Splice's port of the SaltyRTC server taints incoming network bytes with the taint of the sender's IP address. Both SSTP and SaltyRTC are examples of network management applications for which users may need to quickly delete server-side in-memory state for personal safety reasons (e.g., because VPN or



(a) (b) (c)

Figure 1: LFS performance: (a) CPU utilization, (b) average latency per request, and (c) throughput.

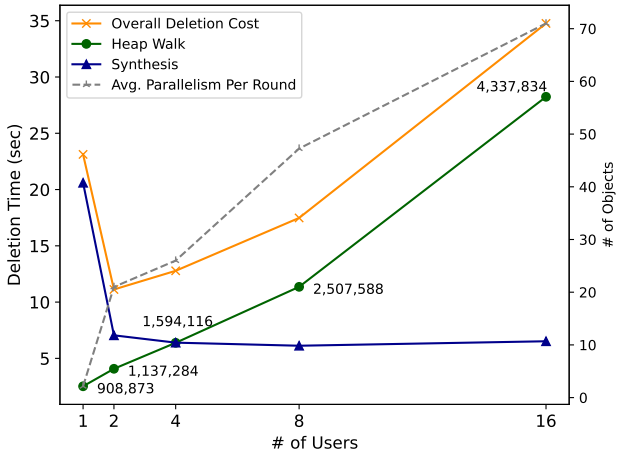


Figure 2: LFS parallel deletion performance.

peer-to-peer network services are illegal in a user's country and a user expects impending scrutiny from law enforcement agencies).

6 EVALUATION

In §5, we demonstrated that porting applications to Splice is straightforward. In this section, we examine the performance of Splice on the ported applications from §5. We study application performance during normal program operation and during user deletion. In all experiments, servers and clients ran on the same 4.6GHz 8-core laptop with 16GB of RAM. We pinned clients and servers to different cores; in some experiments, we injected emulated latency using `netem`. All localhost clients and servers allowed us to effectively set client/server network latency to zero if desired, enabling us to isolate Splice's CPU overheads during normal operation. In each deletion experiment, we scanned memory post-deletion to verify that Splice had correctly excised all of the relevant tainted data.

At a high level, our evaluation shows that Splice provides strong deletion semantics at the cost of a 1.31×–2.69× increase in CPU utilization, and a 1.26×–1.37× increase in memory utilization. Deleting a user's in-memory state on a Splice-aware server takes less than a second for SaltyRPC and SSTP, but up to roughly 70 seconds for our most complex application, a Django-based e-commerce site. We believe that these overheads are reasonable for security-conscious applications; furthermore, these overheads would decrease if our prototype incorporated optimizations to eliminate or parallelize taint tracking instrumentation at run time [33, 42, 68], and parallelize heap walking [66] at deletion time.

6.1 LFS

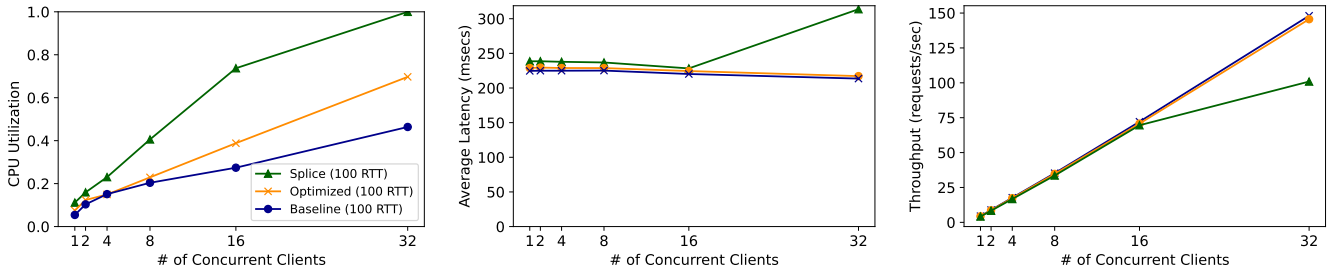
We evaluated our modified Django framework using LFS (Lightning Fast Shop) [18], an e-commerce website that runs atop Django. LFS is an industrial-strength online shopping application that allows customers to browse products, review products, upload payment information, place orders, and so on. We configured Django to use our Redis-style in-memory cache (§5) and a standard PostgreSQL backend.

We evaluated LFS using a workload that mimicked a user who browses multiple items, adds a subset to their cart, and then checks out. In this workload, 90% of the requests are `browse` requests, 8% are `add-to-cart` requests, and 2% are `checkout` requests. We ran a single Django process that was pinned to a single core, and varied the number of clients (each pinned to a single core).

The LFS server used a roughly constant amount of memory, regardless of the number of clients; compared to a baseline (i.e., non-Splice) version of LFS, Splice increased memory usage by roughly 37%. Memory consumption was insensitive to client load because the most common in-memory objects belonged to Django's cache, and the application maintains a cap on the cache's size.

As shown in Fig. 1(a), Splice also generated extra computational overhead due to taint tracking and the maintenance of the type system. However, for an unrealistically fast 0 ms RTT network, the baseline and Splice-enabled LFS both hit CPU saturation at six concurrent clients, causing flatlines in LFS throughput (Fig. 1(c)) and steeper client-perceived increases in LFS response latency (Fig. 1(b)). For a more realistic 100 ms RTT, CPU utilization did not hit 100% for the baseline system or the Splice-enabled one. However, Splice did introduce 31%–59% overhead in CPU utilization. This overhead is comparable to that of other systems that do taint tracking [22] without eliding or parallelizing taint instrumentation [33, 42, 68].

To evaluate Splice's deletion speed, we warmed LFS's in-memory cache with product reviews taken from a real Amazon dataset [32]. We varied the number of users submitting a review from 1 to 16, with each user generating 200 reviews. We then generated a deletion request for a single user.



(a) (b) (c)

Figure 3: MiniTwit performance: (a) CPU utilization, (b) average latency per request, and (c) throughput.

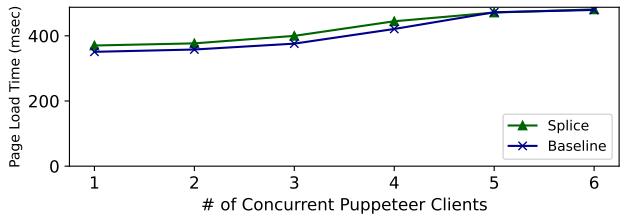


Figure 4: Puppeteer performance of loading stackoverflow.com.

Fig. 2 shows the deletion time as a function of the number of users. We decompose this cost into the time needed to walk the heap, and the time needed to synthesize-delete values. Unsurprisingly, the heap walk time grew linearly with the number of users, because more users result in a larger aggregate number of reviews (and thus more in-memory Python objects). For example, with 16 clients, the heap contained over 4.3 million objects, as shown by the numerical annotation next to the data point in Fig. 2. Our prototype's heap walking implementation is single-threaded, but is amenable to prior techniques that leverage multiple cores to scan the heap [66].

Our prototype does use our novel parallelization scheme for synthesis deletion (§3.4.4). Fig. 2 demonstrates that, for a larger number of users, SPLICE's ability to synthesize in parallel was improved. The reason is that, for a given user to delete (and a given number of in-memory objects belonging to that user), a particular memory value to delete becomes less likely to have a dependence set containing another value from the user to delete (§3.4.4). Introducing just one additional user unlocked many more opportunities for parallel synthesis, as shown by the dashed line, which indicates the average number of tainted objects that can be synthesized in parallel during every round of graph traversal. Even though the opportunity for parallelism increases, the observed synthesis speed flatlined after 4 clients. This is an artifact of our experimental setting, where we have only a limited number of cores to perform the parallel synthesis.

6.2 MiniTwit

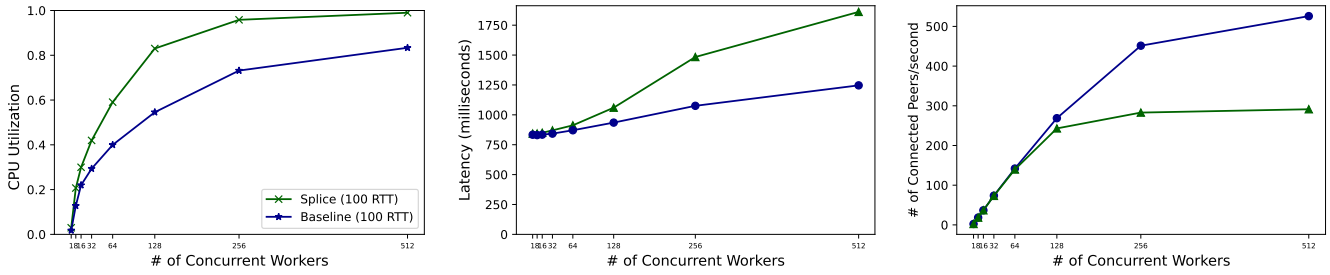
We evaluated MiniTwit on a typical workload containing 95% `view` requests (where a user browses the public timeline of another user), and 5% `tweet` requests (where a user posts a message on their own timeline). We only show results for a realistic network setting (100 ms RTT), while varying the number of concurrent load-generating clients. Fig. 3 shows the CPU utilization, average latency, and throughput of MiniTwit as a function of the number of clients, with and without SPLICE. SPLICE increased memory overhead by roughly 26%, regardless of the number of clients.

SPLICE introduced 54%–169% overhead in CPU utilization as the number of clients increased, causing SPLICE latency and throughput to suffer at 32 clients. These CPU overheads were greater than the ones seen in LFS (§6.1). The reason is that, compared to LFS, MiniTwit performed a large number of string manipulations, resulting in higher object counts and higher overhead induced by SPLICE's taint tracking and type maintenance. The string manipulations were generated by MiniTwit's templating engine, which inserted database results into template strings to create timeline views that were returned to users.

To avoid this overhead, a straightforward (albeit application-specific) optimization is for SPLICE to disable taint tracking within the template engine code; SPLICE would just taint an engine's output string with the union of the taints of the input strings from the database. Fig. 3 shows that this approach dramatically reduced CPU utilization to only 1% - 50%, resulting in SPLICE latency and throughput being comparable to those of the baseline.

To delete a MiniTwit user, we must delete their tweets in PostgreSQL, and their registration information in LevelDB. We focus on the latter, since PostgreSQL's right-to-be-deleted performance is well-understood [62]. As discussed in §3.4.3, LevelDB has both in-memory data and on-disk data. SPLICE must inspect both components to ensure complete deletion.

The in-memory deletion performance followed the same trend as in LFS (Fig. 2): the larger the heap size, the longer it took to walk the heap, while synthesis time was negligible since the amount of registration data for a user was small. For the disk-based structure, deletion overhead grew almost linearly in the number of users: e.g., for 5,000, 10,000, and 100,000 users, the deletion time was 8.5s, 15.7s, and 175.2s. The biggest source of overhead was data deserialization, which occurred when loading the on-disk sstables into memory. Each fully-populated table is ~6.7 MB, takes about 2s to be unmarshalled, and creates approximately 896,000 new heap objects. Once loaded, SPLICE must perform an additional heap walk—for the part of the heap created by the sstable, not the entire heap—to inspect the data.



(a) (b) (c)

Figure 5: SaltyRTC performance: (a) CPU utilization, (b) average latency per peer-to-peer connection, and (c) throughput.

6.3 SSTP VPN

To evaluate the SSTP server, we created a scenario in which one or more web browser clients loaded `stackoverflow.com`. We used Puppeteer instances [17] to generate realistic HTTP traffic, giving each instance two cores. The CPU utilization of the SSTP server remained low throughout the experiment, reaching a maximum of 33% with 6 clients. As shown in Fig. 4, there were minimal client-perceived increases in page load time.

The SSTP server maintained several kinds of OS abstraction objects (§3.1) for each connected client (e.g., a dedicated socket, a pppd child process, and pipes). All of these resources were tainted by the client's IP address, and deleted by the deletion manager. Deletion was fast: regardless of the number of connected clients (we evaluated up to 32 clients), it took less than 5 ms to finish.

6.4 SaltyRTC

SaltyRTC clients only use a SaltyRTC server as a rendezvous point for peer-to-peer conversation; after completing the rendezvous protocol, clients exchange traffic directly. However, the rendezvous handshake does require multiple rounds of communication between the server and the peers that wish to talk to each other. We wrote a traffic generator that provided a configurable steady-state load of N rendezvous attempts. We then scaled N to view Splice's impact on the server's performance.

As shown in Fig. 5(a), Splice introduced 36%–66% overhead in CPU utilization, essentially saturating the server's CPU at a load of 256 concurrent rendezvous attempts. Before that point, Splice's handshaking latency was within 13.4% of the baseline latency (Fig. 5(b)), and server-perceived handshake throughput was within 9.6% of the baseline (Fig. 5(c)). Splice's impact on memory consumption was modest, requiring at worst 18% more memory when handling 512 clients.

The stock SaltyRTC server implements a native deletion function that removes a client's in-memory dictionary state when the client disconnects. However, that function does not remove all references to the now-disconnected client. For example, the client's public key, used as the rendezvous path, remains in several memory locations associated with peers who once communicated with the client. This state of affairs is allowable by the SaltyRTC protocol (which says that a responder *may* remain on an existing path and wait for a new initiator to connect). However, this behavior is bad from the privacy perspective. The Splice-enabled server efficiently deleted all in-memory traces of a disconnected client, requiring less than a second even with 512 concurrent handshakes in flight.

7 RELATED WORK

Taint tracking: Taint tracking is a well-known approach for capturing information flows. The basic idea can be applied at various levels of abstraction. Splice implements it at the managed-language level, similar to TaintDroid [22] and Riverbed [72]. Other systems track data flows at the native code level [8, 23, 59] or at the granularity of OS abstractions like processes and file descriptors [21, 54, 74]. Capturing data flows at the level of POSIX abstractions is too coarse-grained to identify a user's in-memory state with sufficient accuracy. Taint tracking at the native code level sees raw memory writes, and thus could identify per-user data flows to specific memory addresses; however, we eschewed this approach due to the over-tainting that may arise during pointer operations on raw memory [23, 59].

Deleting storage data: Splice focuses on the removal of server-side in-memory data. Other work has investigated how to add GDPR compliance (including the right to be deleted) to SQL storage [37, 60, 64]. This work often employs aggressive per-user partitioning of data to ease the finding and deletion of a particular user's state. For example, in SchengenDB [37], the database tags each data point with the owning user, and ensures that each data point is stored in exactly one place (i.e., exactly one row). By disallowing duplicates or derivatives of that data, SchengenDB makes enumerating and deleting a user's data straightforward, but limits its schema expressivity, and forfeits the performance gains that arise from storing duplicate or derived data. Splice strives to work with arbitrary data structures, and thus cannot prohibit duplication or derivation. However, Splice does require data structures to be annotated with symbolic constraints.

Facebook uses the Delf framework [11] to orchestrate deletion across the multiple SQL databases, key/value stores, and graph stores that comprise Facebook's backend. When developers introduce a new data type, Delf requires developers to provide object annotations (which inform Delf when to delete objects, e.g., after a TTL expiration) and edge annotations (which determine how Delf handles objects that are reachable from a now-deleted object). This approach makes sense for the more structured world of backend storage, but less sense for in-memory data structures which do not

use standardized, formal notions like primary keys and foreign keys to connect various objects to each other. Web developers are already familiar with defensive programming (§1), so Splice leverages this familiarity to make it easier to reason about deletion semantics. Splice also reduces annotation burdens by pre-annotating data structures in a language's standard library. Another key difference between Delf and Splice is that Delf handles a deletion request asynchronously. In contrast, an important goal of Splice is to handle a request as soon as a quiescent period arrives, which will typically be within hundreds of milliseconds of the request arrival.

In the context of the GDPR's right to deletion, Shah et al. looked at Redis, a popular key-value store often used by web services [61]. Redis provides built-in support for synchronous key deletion, as well as lazy deletion in which Redis deletes a key at some point after the key's expiration period has elapsed. Shah et al. found that Redis's lazy deletion is quite lazy, sometimes requiring hours to delete an expired key. Both lazy and synchronous deletion also left remnants of a key/value pair in Redis's on-disk operation log. The Splice version of the LevelDB key/value store logically treats the store's on-disk data structures as extensions of the in-memory data structures (§3.4.3); thus, at deletion time, Splice walks heap data that resides in both memory and persistent storage, synthesize-deleting the necessary values.

Encryption-based deletion: Vanish [27] and CleanOS [67] keep sensitive data encrypted at rest. "Deletion" occurs via the discarding of the associated encryption keys (which renders the encrypted data useless). As mentioned in §1, even systems that use at-rest data encryption often store cleartext in-memory data that is influenced by user activity. Splice provides a way to find and delete such values.

8 CONCLUSION

Splice is a framework that helps applications to locate and delete the in-memory state belonging to a particular user. Splice achieves this goal through a novel combination of taint tracking, a defensive programming model enabled by a new type system, and deletion by synthesis. We ported real applications to Splice, and showed that the porting burdens are low, and that Splice's performance overheads are reasonable for security-conscious applications.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was partially supported by the NSF under grant CNS-2245442. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Accessed 19th September 2023. pppd. https://linux.die.net/man/8/pppd.
[2] Accessed 19th September 2023. Signal. https://signal.org.
[3] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2008. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of IEEE Symposium on Security and Privacy* (Oakland, CA). 387–401.
[4] Austen Barker, Yash Gupta, Sabrina Au, Eugene Chou, Ethan L. Miller, and Darrell D. E. Long. 2020. Artifice: Data in Disguise. In *Proceedings of the Conference on Mass Storage Systems and Technologies* (Santa Clara, CA).
[5] Josiah L. Carlson. 2013. *Redis in Action.* Manning Publications Co.
[6] Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. 2014. Heartbleed 101. *IEEE Security & Privacy* 12, 4 (2014), 63–67.
[7] Shell Chan. 2021. SSTP Server. https://github.com/sorz/sstp-server.
[8] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the IEEE Symposium on Computers and Communications* (Cagliari, Sardinia, Italy). 749–754.
[9] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. 2005. Shredding Your Garbage: Reducing Data Lifetime through Secure Deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium* (Baltimore, MD).
[10] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully Concurrent Garbage Collection of Actors on Many-core Machines. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, IN). 553–570.
[11] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. 2020. DELF: Safeguarding Deletion Correctness in Online Social Networks. In *29th USENIX Security Symposium* (Boston, MA).
[12] Juan José Conti and Alejandro Russo. 2010. A Taint Mode for Python via a Library. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems* (Espoo, Finland). 210–222.
[13] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems* 25, 2 (June 2000), 179–227.
[14] Alexei Czeskis, David J. St. Hilaire, Karl Koscher, Steven D. Gribble, Tadayoshi Kohno, and Bruce Schneier. 2008. Defeating Encrypted and Deniable File Systems: TrueCrypt v5.1a and the Case of the Tattling OS and Applications. In *Proceedings of the USENIX Workshop on Hot Topics in Security* (San Jose, CA).
[15] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.
[16] Jeff Dean and Sanjay Ghemawat. 2011. LevelDB: A Fast Persistent Key-Value Store. https://opensource.googleblog.com/2011/07/leveldb-fast-persistent-key-value-store.html.
[17] Chrome DevTools. 2021. Puppeteer: Headless Chrome Node.js API. https://github.com/puppeteer/puppeteer.
[18] Kai Diefenbach. 2021. LFS: Lightening Fast Shop. https://github.com/diefenbach/django-lfs.
[19] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA). 303–320.
[20] Django Software Foundation. 2023. Security in Django. https://docs.djangoproject.com/en/4.2/topics/security/.
[21] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. 2016. Thoth: Comprehensive Policy Compliance in Data Retrieval Systems. In *Proceedings of the 25th USENIX Security Symposium* (Austin, TX). 637–654.
[22] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2, Article 5 (June 2014), 5:1–5:29 pages.
[23] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, L Fowler, and Murphy McCauley. 2010. Towards Practical Taint Tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92* (2010).
[24] ExpressVPN. 2023. ExpressVPN: High-speed, Secure, and Anonymous VPN Service. https://www.expressvpn.com.
[25] Jeff Forcier, Paul Bissex, and Wesley J Chun. 2008. *Python Web Development with Django.* Addison-Wesley Professional.
[26] Yu Gao, Wensheng Do, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed System. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL). 539–550.
[27] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. 2009. Vanish: Increasing Data Privacy with Self-Destructing Data. In *Proceedings of the 18th USENIX Security Symposium* (Montreal, Canada). 299–316.
[28] Lennart Grahl. 2015. SaltyRTC, an End-to-end Encrypted Signalling Protocal. https://saltyrtc.org.
[29] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Clara, CA). 1–16.
[30] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure Recovery: When the Cure Is Worse Than the Disease. In *14th Workshop on Hot Topics in Operating Systems* (Santa Ana Pueblo, NM).
[31] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. 2006. A Classification of SQL-injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Vol. 1. 13–15.

[32] Ruining He and Julian J. McAuley. 2016. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *Proceedings of the International Conference on World Wide Web* (Montreal, Canada). 507–517.

[33] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *ACM SIGSAC Conference on Computer and Communications Security* (Berlin, Germany). 235–246.

[34] Richard Jones and Rafael Lins. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, Inc.

[35] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit flows: Can't live with 'Em, can't live without 'Em. In *Proceedings of the 4th International Conference on Information Systems Security* (Hyderabad, India). 56–70.

[36] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley.

[37] Tim Kraska, Michael Stonebraker, Michael Brodie, Sacha Servan-Schreiber, and Daniel Weitzner. 2019. SchengenDB: A Data Protection Database Proposal. In *Proceedings of the VLDB Workshop on Heterogeneous Data Management, Polystores, and Analytics for Healthcare* (Los Angeles, CA). 24–38.

[38] Chen Luo and Michael J Carey. 2020. LSM-based Storage Techniques: a Survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[39] Dan Luu and Yao Yue. 2022. A decade of major cache incidents at Twitter. https://danluu.com/cache-incidents/.

[40] Nick Mathewson, Azat Khuzhin, and Niels Provos. 2017. libevent – an event notification library. https://libevent.org.

[41] Microsoft. 2021. MS-SSTP: Secure Socket Tunneling Protocol (SSTP). https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-sstp/.

[42] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. 2015. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proceedings of the 24th USENIX Security Symposium* (Washington, D.C.). 65–80.

[43] MITRE Corporation. 2022. CVE-2022-41318. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-41318.

[44] MITRE Corporation. 2022. CVE-2022-42905. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-42905.

[45] MITRE Corporation. 2023. CVE-2022-46143. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-46143.

[46] MITRE Corporation. 2023. CVE-2023-24565. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-24565.

[47] Moni Naor and Vanessa Teague. 2001. Anti-persistence: History Independent Data Structures. In *Proceedings of the ACM symposium on Theory of computing* (Hersonissos, Greece). 492–501.

[48] Sverker Nilsson and YiFei Zhu. 2019. Guppy 3: A Python Programming Environment and Heap Analysis Toolset. https://zhuyifei1999.github.io/guppy3/.

[49] Nord Security. 2023. NordVPN: The best online VPN service for speed and security. https://nordvpn.com/.

[50] Tavis Ormandy. 2017. Issue 1139: cloudflare: Cloudflare Reverse Proxies are Dumping Uninitialized Memory. https://bugs.chromium.org/p/project-zero/issues/detail?id=1139.

[51] OWASP Foundation. 2023. Web Application Security Testing: Input Validation Testing. https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/07-Input_Validation_Testing/README.

[52] Pallets. 2010. Flask. https://flask.palletsprojects.com/en/2.0.x/.

[53] Pallets. 2016. MiniTwit. https://github.com/pallets/flask/tree/0.12.x/examples/minitwit.

[54] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical Whole-System Provenance Capture. In *Proceedings of the Symposium on Cloud Computing* (Santa Clara, CA). 405–418.

[55] Timothy Peters, Mark A. Gondree, and Zachary N. J. Peterson. 2015. DEFY: A Deniable, Encrypted File System for Log-Structured Storage. In *Proceedings of the 22^{nd} Network and Distributed System Security Symposium* (San Diego, CA).

[56] Manoj Plakal and Charles N Fischer. 2000. Concurrent Garbage Collection Using Program Slices on Multithreaded Processors. In *Proceedings of the International Symposium on Memory Management* (Minneapolis, MN). 94–100.

[57] Andrei Sabelfeld and Alejandro Russo. 2009. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In *Proceedings of the International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics* (Novosibirsk, Russia). 352–365.

[58] Theodoor Scholte, William Robertson, Davide Balzarotti, and Engin Kirda. 2012. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. In *Proceedings of the ACM Symposium on Applied Computing* (Trento, Italy). 1419–1426.

[59] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy* (Berkeley/Oakland, CA). 317–331.

[60] Malte Schwarzkopf, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2019. Position: GDPR Compliance by Construction. In *Proceedings of the VLDB Workshop on Heterogeneous Data Management, Polystores, and Analytics for Healthcare* (Los Angeles, CA). 39–53.

[61] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. 2019. Analyzing the Impact of GDPR on Storage Systems. In *Proceedings of the 11^{th} USENIX Conference on Hot Topics in Storage and File Systems* (Renton, WA).

[62] Supreeth Shastri, Vinay Banakar, Melissa Wasserman, Arun Kumar, and Vijay Chidambaram. 2020. Understanding and Benchmarking the Impact of GDPR on Database Systems. *Proceedings of VLDB Endowment* (March 2020), 1064–1077.

[63] Facebook Open Source. 2021. RocksDB: A persistent key-value store for fast storage environments. http://rocksdb.org.

[64] Jonas Spenger, Paris Carbone, and Philipp Haller. 2021. Pods: Privacy Compliant Scalable Decentralized Data Services. In *Proceedings of the VLDB Workshop on Polystore Systems for Heterogeneous Data in Multiple Databases with Privacy and Security Assurances* (Copenhagen, Denmark). 70–82.

[65] M. Stigge, H. Plotz, W. Muller, and J.P. Redlich. 2006. Reversing CRC—Theory and Practice. HU Berlin Public Report: SAR-PR-2006-05.

[66] Kun Suo, Jia Rao, Hong Jiang, and Witawas Srisa-an. 2018. Characterizing and Optimizing Hotspot Parallel Garbage Collection on Multicore Systems. In *Proceedings of the 13^{th} EuroSys Conference* (Porto, Portugal). 35:1–35:15.

[67] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *10th USENIX Symposium on Operating Systems Design and Implementation* (Hollywood, CA). 77–91.

[68] Zhiyou Tian, Cong Sun, Dongrui Zeng, and Gang Tan. 2023. podft: On Accelerating Dynamic Taint Analysis with Precise Path Optimization. In *Proceedings of NDSS Workshop on Binary Analysis Research* (San Diego, CA).

[69] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *Proceedings of the 29th USENIX Security Symposium* (Boston, MA). 505–522.

[70] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25^{th} Symposium on Operating Systems Principles* (Monterey, CA). 137–152.

[71] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium* (San Diego, CA).

[72] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-Defined Privacy Constraints in Distributed Web Services. In *Proceedings of the 16^{th} USENIX Conference on Networked Systems Design and Implementation* (Boston, MA). 615–630.

[73] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14^{th} ACM Conference on Computer and Communications Security* (Alexandria, VA). 116–127.

[74] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing Distributed Systems with Information Flow Control. In *Proceedings of the 5^{th} USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, CA). 293–308.

[75] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. 2020. When is the Cache Warm? Manufacturing a Rule of Thumb. In *Proceedings of the 12^{th} USENIX Workshop on Hot Topics in Cloud Computing.*

[76] David Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *ACM SIGOPS Operating Systems Review* 45, 1 (2011), 142–154.