LeakLess: Selective Data Protection Against Memory Leakage Attacks for Serverless Platforms

Maryam Rostamipoor
Stony Brook University
mrostamipoor@cs.stonybrook.edu

Seyedhamed Ghavamnia
University of Connecticut
sghavamnia@uconn.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

***Abstract*—As the use of language-level sandboxing for running untrusted code grows, the risks associated with memory disclosure vulnerabilities and transient execution attacks become increasingly significant. Besides the execution of untrusted JavaScript or WebAssembly code in web browsers, serverless environments have also started relying on language-level isolation to improve scalability by running multiple *functions* from different customers within a single process. Web browsers have adopted process-level sandboxing to mitigate memory leakage attacks, but this solution is not applicable in serverless environments, as running each function as a separate process would negate the performance benefits of language-level isolation.**

In this paper we present LeakLess, a selective data protection approach for serverless computing platforms. LeakLess alleviates the limitations of previous selective data protection techniques by combining in-memory encryption with a separate I/O module to enable the safe transmission of the protected data between serverless functions and external hosts. We implemented LeakLess on top of the Spin serverless platform, and evaluated it with real-world serverless applications. Our results demonstrate that LeakLess offers robust protection while incurring a minor throughput decrease under stress-testing conditions of up to 2.8% when the I/O module runs on a different host than the Spin runtime, and up to 8.5% when it runs on the same host.

I. Introduction

Serverless computing [1] is an emerging cloud-based application deployment model that abstracts server management from application code. The key component of serverless platforms, such as AWS Lambda [2] and Cloudflare Workers [3], is a Function-as-a-Service (FaaS) architecture, complemented by integrated database and storage services. FaaS enables developers to focus on application logic by splitting it into smaller *functions*, without worrying about operational issues such as load balancing and scaling.

Traditionally, FaaS platforms [2], [4], [5] leverage hardware or OS-level virtualization to isolate different tenants' functions into different containers or VMs, due to their strong isolation guarantees. To improve scalability, however, serverless platforms by Cloudflare [3], Fastly [6], and others [7], [8] have gradually started leveraging language-level isolation based on the JavaScript V8 engine [9] or the WebAssembly [10] sandbox for running multiple functions within a single process. Language-level isolation improves efficiency, enabling the execution of orders of magnitude more functions with the same number of processes and amount of hardware resources. At the same time, however, the weaker language-level isolation increases the risk of cross-tenant data leakage attacks.

Language-level sandboxing protects against memory disclosure vulnerabilities due to the memory safety guarantees of Javascript and WebAssembly. Still, bugs in the language runtime itself uphold the threat of memory leakage attacks, as exemplified by the continuous discovery of out-of-bounds memory access vulnerabilities in the V8 engine [11], [12] and WebAssembly runtimes [13]–[15]. Researchers have explored various methods to ensure the accuracy of memory isolation checks in language-level sandboxes, by verifying the safety of generated binaries [16], building verified compilers [17], and fuzzing compilers [18].

In addition to memory disclosure vulnerabilities, the threat of data leakage has been exacerbated by the recent spate of transient execution attacks [19], [20], which can leak otherwise inaccessible data from memory through residual microarchitectural side effects. Transient execution attacks do not violate memory boundaries or data flows enforced by the process itself [19], making bounds checking, software fault isolation, and even memory-safe languages inadequate for protecting against them [21]. In FaaS platforms, a malicious tenant can mount Spectre-like attacks to leak secrets from other tenants. As an example, Schwarzl et al. [22] demonstrated how a malicious function can leak sensitive data from a co-located function in Cloudflare Workers [3] using a Spectre attack.

Serverless functions are a prime target for transient execution attacks because they typically contain sensitive data as part of their code, or are used to handle sensitive user-provided data. For example, functions often contain secret keys for making authenticated requests to external APIs and services, or handle private user data such as passwords and credit card numbers. Serverless platforms even provide specialized "secrets" management services [23], [24]. Cloudflare recently announced that about three million Workers scripts reference sensitive data through the Secrets Store API [24].

Defending against transient execution attacks in FaaS platforms is an open problem, with vendors deploying half-measures, such as disallowing timer APIs and periodically

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230035
www.ndss-symposium.org

shuffling the whole memory [25]—full process isolation is not applicable in this context, as running each workload in a separate process would defeat the purpose of language-based isolation. DyPrIs [22] does use process isolation to migrate any suspiciously behaving functions into separate processes, but the approach relies on anomaly detection, which suffers from false positives and false negatives. Crucially, process-level isolation is not a robust solution in the first place, as transient execution attacks can leak secrets across processes [26]–[30] and even SGX enclaves [31]. Swivel [32] uses a compiler-based approach to harden WebAssembly modules against certain types of Spectre attacks, but the numerous introduced runtime checks incur a very high runtime overhead.

In this paper we propose *LeakLess*, a practical approach for countering data leakage attacks in serverless platforms, which relies on selective in-memory encryption of developer-annotated sensitive data. Instead of trying to prevent the myriad ways in which the CPU can be tricked into disclosing sensitive data from memory (of the same or a different process), LeakLess achieves future-proof protection by accepting the fact that sensitive data may be leaked, and ensuring that it will remain useless for the attacker—any leaked data from memory will always be encrypted.

Prior works on selective data encryption [33], [34] and isolation [35], [36] are not applicable in this setting, as they are designed to protect only *internal* application data. LeakLess introduces a separate I/O module that mediates communication between external parties and the serverless runtime, and handles all cryptographic operations for transmitting or receiving the plaintext version of the sensitive data. The I/O module runs as a separate process either on the same host, offering protection against typical *intra-process* data leakage attacks due to i) out-of-bounds memory disclosure vulnerabilities in the runtime or ii) transient execution attacks, or on a separate VM or physical host, offering additional protection against *cross-process* [26]–[31], [37], [38] and *user-to-kernel* [39]–[43] attacks.

We implemented LeakLess on top of the Spin serverless platform [44], which uses WebAssembly to run different functions in the same language runtime. To assess the practical applicability and compatibility of LeakLess with real-world environments, we collected 1,074 publicly available serverless applications from public repositories. Based on our assessment, nearly half of them (449) contain at least one type of sensitive data, including authentication secrets, request signing keys, database credentials, and user passwords [45]–[47].

Our current implementation fully supports immutable data, as well as certain types of mutable data that undergoes common cryptographic operations, such as JSON web token (JWT) signing and verification. Among the 449 applications that contain sensitive data, LeakLess fully supports 407 of them (91%), and partially supports 33 (7%), i.e., they contain at least one secret that LeakLess can protect. We experimentally evaluated our prototype implementation with six real-world and widely-used serverless applications, and demonstrate its effectiveness in preventing memory disclosure and transient execution attacks with minimal performance impact.

In summary, we make the following main contributions:

- We propose LeakLess, a software architecture for selective data protection in serverless platforms, which offers future-proof protection against memory disclosure and transient execution attacks.
- We alleviate the limitations of previous selective data protection approaches by combining in-memory encryption with a separate I/O module to enable the safe transmission of the protected data.
- We implemented LeakLess on top of Spin [44], a serverless platform which relies on WebAssembly to run functions written in various languages.
- We collected a set of 1,074 real-world serverless applications and provide a thorough analysis of the types of sensitive data handled by them.
- We experimentally evaluated LeakLess with micro-benchmarks and six real-world applications, and demonstrate that it offers robust protection of secrets while incurring a service throughput reduction of up to 2.8% when the I/O module runs on a different host, and up to 8.5% when it runs on the same host as the Spin runtime, under stress-testing conditions.

Our implementation is publicly available as an open-source project at https://github.com/mrostamipoor/LeakLess.

II. Background and Motivation

A. WebAssembly

WebAssembly (Wasm) [48] is a portable binary instruction format for a stack-based virtual machine. As a portable compilation target, it allows programs written in different languages (e.g., C, C++, and Rust) to be compiled directly into Wasm. Wasm employs a linear memory model, structured as a resizable byte array as its main storage. Access to this memory is facilitated through specific `load` and `store` instructions, with runtime bounds checks to ensure safety. Furthermore, Wasm enforces strict control flow rules, effectively preventing jumps to arbitrary memory locations. These mechanisms enable Wasm runtimes [49], [50] to securely sandbox untrusted third-party code within applications.

While WebAssembly was originally designed to be run within web browsers, it has also become popular for server-side applications. One notable example is Wasmtime [49], an optimizing runtime for WebAssembly, designed to execute Wasm programs either standalone or embedded in other applications. Wasmtime leverages the WebAssembly System Interface (WASI) [51], which augments Wasm's capabilities by providing access to essential operating system functionalities like file and socket management.

B. Function-as-a-Service (FaaS) Platforms

Due to the large memory footprint and increased cold start latency of VMs and containers when used in FaaS platforms, language-based isolation [6], [44], [50], [52]–[54] has recently emerged as a compelling alternative, and is already used by popular platforms such as Cloudflare Workers [3] and Fastly [6]. In these environments, multiple functions by different customers run within the same runtime instance, each isolated inside a sandboxed execution environment [3], [6], [54]. However, this isolation can be weakened due to bugs in the language runtime [11]–[15], leading to out-of-bounds read capabilities by malicious functions.

Even if we assume that the memory isolation guarantees of Wasm effectively protect sensitive data within a function from memory-related errors, this assumption does not hold true for transient execution attacks such as Spectre [19], [55]. Given that functions from different customers run within the same process, a malicious function can mount Spectre-like attacks to leak sensitive data from other co-located functions [22].

Sensitive data include user-provided information such as passwords or credit card numbers, as well as developer-provided secrets required for the operation of the application. The latter are prevalent in serverless functions, which often interact with various types of back-end services, including traditional web servers, platform as a service providers [4], [56], other serverless platforms [4], [57], and storage services [58]. These services often use API keys for authentication and access control, which are typically represented as immutable strings set into custom headers in API calls. Additionally, some platforms contain various types of secrets that are involved in certain computational processes. These include secret keys used data signing or JWT token verification. Functions must retain these keys in memory to use them when communicating with third-party services, making them a lucrative target for data leakage attacks. Using LeakLess, developers can simply annotate these keys as *sensitive*, ensuring that they will always remain encrypted in the address space of the Wasm runtime.

C. Transient Execution Attacks

Transient execution attacks, such as Spectre [19] and Meltdown [39], rely on exception or branch misprediction events that have an effect on the CPU's microarchitectural state [20]. Spectre attacks have various forms, with the first two variants being the most well known. Variant 1 (Spectre-BCB), enables speculative out-of-bound memory access. Variant 2 (Spectre-BTI) takes advantage of unconditional indirect branches by forcing the speculative execution of an incorrect branch target. This is achieved by manipulating the branch target buffer with attacker-controlled targets. This poisoning is possible across different address spaces [19] and privilege boundaries [41].

In FaaS platforms that rely on language-based isolation, an attacker can mount Spectre attacks to leak another tenant's sensitive data. To prevent them, platforms like Cloudflare [3] disable all known timers and primitives that could be abused to build timers. Moreover, low-level instructions that are used in Spectre attacks are not exposed to Wasm code. Schwarzl et al. [22], however, showed that attackers can still leak secrets from co-located functions. Despite the lack of a precise timer, attackers can rely on amplification techniques and leveraging a remote timing server, while other works have shown that side-channel attacks can be performed without the need for an architectural timer at all [59]–[61].

III. Threat Model

Our threat model includes data leakage threats due to both memory disclosure vulnerabilities and transient execution attacks, which give adversaries the capability to read (i.e., leak) arbitrary user-space and kernel-space memory and access confidential data from the functions of other tenants. We assume that the attacker is limited to just achieving data leakage (instead of arbitrary code execution in the context of a different function), either because of the nature of the vulnerability, or due to the presence of mitigations against arbitrary code execution. The attacker can be a legitimate user of the FaaS platform, who can upload function code written in any supported language, which the platform will then compile and execute alongside other users' functions.

Besides software memory disclosure vulnerabilities in the serverless platform runtime, data leakage through transient execution attacks is the main motivation of our work. *Intra-process* attacks [19] bypass memory safety to leak data from the same process. *Cross-process* attacks [26]–[30], [37], [38], [62], [63] bypass process-level isolation to leak data from different processes (or even enclaves [31]). *User-to-kernel* attacks [39]–[42] mounted from user space bypass privilege levels and leak data from the kernel or different processes (e.g., through `physmap` [64]).

LeakLess offers flexibility in defending against all above attacks, by choosing the appropriate isolation boundary between the serverless runtime and the I/O module for the desired threat model. If only same-process attacks are considered, then the I/O module can run on the same host; if cross-process and user-to-kernel attacks are considered, then the I/O module can run on a separate virtual machine; if cross-VM attacks are considered, then the I/O module can run on a separate physical host. Inadvertent data exposure through computation that impacts control flow, e.g., side channels that measure execution timing, is outside our threat model.

IV. Design

LeakLess uses a combination of in-memory encryption and I/O brokering to protect sensitive data in Function-as-a-Service (FaaS) platforms against memory disclosure vulnerabilities and transient execution attacks. Prior works on selective data encryption [33], [34] have demonstrated that by always keeping sensitive data encrypted in memory, any attempt to leak protected data through an arbitrary memory access capability or microarchitectural side channel will effectively fail, as the leaked data will always remain encrypted.

Although these approaches work well for internal application data (e.g., private keys), they are not adequate for protecting sensitive data that needs to cross the confines of the application, as the data has to first be decrypted—at which point it becomes vulnerable to leakage attacks. In serverless applications, outgoing and incoming data is precisely the type of sensitive data that needs protection. Due to their stateless nature, serverless functions cannot retain information between invocations, and thus rely on platform-provided or external services to store, retrieve, and update data. Therefore, developer-provided secrets

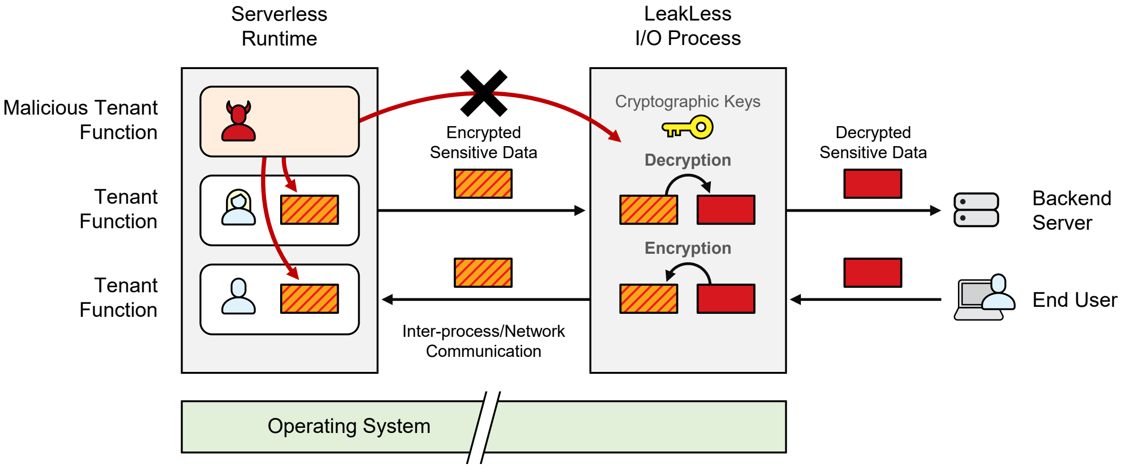


Fig. 1: LeakLess protects sensitive data in FaaS platforms by always keeping the data encrypted in memory and managing its cross-process flow. Cryptographic operations are handled by a separate I/O module, and thus the secret keys and plaintext version of the data are never exposed in the memory of the serverless runtime. A malicious tenant can still leak the sensitive data, but only in its encrypted form, without having access to the key. The I/O module can transparently run on a separate VM or even physical host to offer additional protection against cross-process and user-to-kernel transient execution attacks.

(e.g., external API keys or authentication tokens) are routinely transmitted to back-end or third-party services. Sensitive client-provided data (e.g., credit card numbers) must also be protected once entering the address space of a serverless function.

LeakLess addresses this challenge by introducing a dedicated I/O module running as a separate process that encrypts (or decrypts) any sensitive incoming (or outgoing) data to (or from) a serverless function, as shown in Figure 1. The main runtime process is responsible for running serverless functions that may handle sensitive data. Any attempt of a malicious function to leak protected data from another function within the same address space will still succeed, but the leaked data will remain encrypted—and thus useless for the attacker, because the cryptographic keys *are not present* in the address space of the serverless runtime at all, and thus they cannot be leaked.

The I/O module acts as a proxy between LeakLess-protected functions and back-end services or end users, and decrypts or encrypts any sensitive data before forwarding it to the remote party or the function. The I/O module does have access to the encryption keys, and the plaintext form of the protected data is exposed in its address space, but this does not pose any security risk because the I/O module does not run any tenant-supplied code. This is not the case for *cross-process* transient execution attacks, which can be addressed by running the I/O module on a separate VM or physical host, as we discuss in Section IV-D.

A key goal of LeakLess is to provide practical protection without burdening developers, and thus it requires minimal application changes. All handling of protected data is performed transparently by the language runtime and the I/O module. Developers just need to annotate in the function source code or application configuration file (and if necessary, at the client-side application logic) any variables that contain sensitive data. For the rest of this section, we assume a FaaS platform that relies on WebAssembly modules to run tenants' functions.

A. *Sensitive Data Annotation and Handling*

We distinguish between two main types of sensitive data: i) *internal data*, which originates from within a function, and is either used only internally by the function, or may be transmitted to external entities; and ii) *external data*, which originates from external entities (e.g., end users or backend servers) and is received by a function.

1) Internal Data Annotation: In FaaS platforms, secrets are typically defined through one of the following methods: i) hard-coded directly in the function's source code, or ii) specified in configuration files or through dedicated secret management APIs. LeakLess supports the annotation of sensitive data for both of these methods.

Source Code Annotation: For sensitive data hard-coded in the source code, developers must annotate the respective variable's definition. Listings 1 and 2 present examples in Rust and Go, respectively. The variable AUTH_TOKEN (line 2) holds a secret authorization token for an external API, which can be protected by simply annotating it as "secret" (line 1).

Listing 1: Simplified Rust code for sending a request with sensitive data (an authorization token) to a backend server.

```
1  #[LEAKLESS_SECRET]
2  const AUTH_TOKEN: &str = "secret-token";
3  let mut res =
4  spin_sdk::outbound_http::send_request(
5    http::Request::builder()
6      .method("GET")
7      .header("Authorization",
8       HeaderValue::from_str(AUTH_TOKEN)?)
9      .uri("https://backend.com/image.jpg")
10     .body(None)?,
11 )?;
```

In both cases, the first line of code conveys that the following line defines sensitive data. For Go, before compilation, the go generate directive is used to scan for the special LeakLess comment. In both languages, while parsing the second line, the compiler or the directive fetches the variable's name and value and replaces it with the encrypted value. Our notation is inspired by the data annotations used for model validation in the ASP.NET MVC framework [65].

Listing 2: Simplified Go code for sending a request with sensitive data (an authorization token) to a backend server.

```go
//#[LEAKLESS_SECRET]
const AUTH_TOKEN = "secret-token"
req, _ := http.NewRequest("GET", "https://backend.com/image.jpg", nil)
req.Header.Add("Authorization", "Bearer "+AUTH_TOKEN)
resp, _ := spinhttp.Send(req)
```

Language-agnostic Annotation: Hard-coding secrets in the source code increases the chances of inadvertent data leakage (e.g., through committing code to public repositories). Instead, many FaaS platforms encourage developers to define secrets through configuration files or securely store them through "secrets" management APIs. To support this category of sensitive data, LeakLess provides a language-agnostic annotation. This is achieved through a simple extension to the *existing* specifications for defining sensitive variables in configuration files or secrets APIs. In the example of Listing 3, the variable Authentication_Token (line 2) is defined in a configuration file, and also has the platform-supported secret option enabled, which specifies that it should be treated as sensitive data. This protects it from unauthorized access, exposure in logs, and accidental disclosure in code repositories, but the variable will still exist in memory as plaintext, remaining vulnerable to transient execution attacks.

To protect it with LeakLess, developers can simply provide the additional leakless_secret option in the configuration file to annotate it as a LeakLess secret, while keeping the rest of the code intact. The bottom of Listing 3 shows how this variable is then retrieved in a function's body in Rust and Go, respectively (the rest of the code would be similar to Listing 1 and Listing 2).

Listing 3: Example of defining a sensitive variable in a configuration file (top) and using it in the code (middle/bottom).

```
1 // Annotation of secrets in configuration file
2 Authentication_Token = {default = "secret-token", secret = true, leakless_secret = true}
3 // Rust code
4 let authToken = config::get("Authentication_Token")
5      .expect("could not get variable");
6 // Go code
7 authToken, err := config.Get("Authentication_Token")
```

Cross-process Flow of Sensitive data: When the Serverless function containing the code of Listing 1 (and similarly of Listing 3), is triggered by an HTTP request coming from an end user, the function is executed and sends an HTTP GET request. Listing 4 shows the generated HTTP GET request.

Listing 4: Request generated by the code of Listings 1 or 2.

```http
GET /image.jpg HTTP/1.1
Host: backend.com
Authorization:
LEAKLESS_AfvUK704aQFsjnTCyfCTrA==
```

Instead of the plaintext value of the authorization token, the "Authorization" HTTP header contains the encrypted value of the token, prepended with the prefix LEAKLESS_ (which denotes that the following data is encrypted[1]). When the I/O module encounters such a sensitive value as part of an outgoing request, it decrypts the value and replaces it with the plaintext form of the data, before forwarding it to the actual recipient.

2) *External Sensitive Data Annotation:* External data is handled similarly, but in the opposite direction. The developer just needs to ensure that any client-supplied sensitive data is distinguished from the rest of the data by marshaling it with the LEAKLESS_ prefix. For the credit card number example, the JavaScript code running in the user's browser annotates the user-supplied credit card number before transmitting it to the function, as shown in Listing 5.

Listing 5: A client-originating HTTP POST request containing two annotated (plaintext) sensitive values.

```http
POST /updateInfo HTTP/1.1
Host: framework-domain
Authorization: LEAKLESS_HES6ZRVmB7fkLtd1Z
Content-Type: application/json

{
 "cardNumber": "LEAKLESS_374245455400126",
 "description": "CreditCardInfo"
}
```

In this example, the browser sends a POST request containing sensitive data in both the HTTP headers (Authorization) and in the request body (cardNumber). Once received by the I/O module, both values will be recognized as sensitive due to their LeakLess-specific marshaling, and will be encrypted before the request is forwarded to the serverless function.

A critical detail here is that the encrypted values will still maintain the LEAKLESS_ prefix. This design supports the transparent flow of sensitive data in and out of a function. Continuing with the same example, the function typically will have to send the user-supplied credit card number to a third-party payment processing service through an authenticated API. The function's code is oblivious to the fact that the credit card number is marshaled (i.e., encrypted), and will handle it in the same way as before. When the request towards the payment processor is received by the I/O module, the LEAKLESS_ prefix will still be present, and thus the I/O module will decrypt the value and transmit the plaintext version of the credit card number to the payment processor.

[1] For illustration purposes only. Proper marshaling with a sufficiently long, randomly generated prefix and suffix is a more appropriate choice.

B. Supported Types of Sensitive Data and Operations

LeakLess currently supports only immutable data, i.e., we assume that sensitive data will not be altered by a function. This is not a limitation of our design, but of our current implementation. More complex computations on sensitive data can be performed by employing register-based encryption [34]. Despite this limitation, some common operations can still be performed, including comparing encrypted values and concatenating encrypted with non-encrypted strings. This enables LeakLess to support the majority of real-world applications that handle sensitive data, as discussed in Section VI-A.

To expand the range of supported applications even further, LeakLess also supports some common cryptographic operations, such as signing and verification, which are often performed by serverless functions when communicating with external services. While most back-end services use API keys and OAuth tokens for authentication (which LeakLess directly supports), others, such as the above object storage services, require request signing with a secret key as part of the authentication process. Another common case involves verifying or signing JWT tokens for end users. However, if the relevant secret keys are encrypted by LeakLess, verification and signing would not be able to proceed.

To support these common scenarios, we have outsourced these operations to the I/O module, which *does* have access to the secret keys used for verification or signing. Developers just need to specify the type of operation that must be performed in a configuration file, as commonly done in cloud platforms for other operations as well. For example, in Cloudflare Workers, developers similarly identify external services and secrets required for their Workers as part of a configuration file or through environment variables [66].

Listing 6: Example of enabling signing by the I/O module for an outgoing Amazon S3 request (top) and enabling verification by the I/O module for a JWT token (bottom).

```
//Annotation for signing requests
S3_sign_key = { default = "7IhhnziifKKdcf0",
    leakless_secret = true, leakless_operation =
    "request-sign" }

//Annotation for verifying JWT tokens
JWT_secret = { default = "secret_key",
    leakless_secret = true, leakless_operation =
    "verify-jwt" }
```

When the `leakless_operation` attribute is defined for a LeakLess-protected secret (Listing 6), the I/O module performs the requested operation on behalf of the function. In this example, the function communicates with Amazon S3, and the developer has specified that the secret is used for request signing. When the I/O module receives the request from the function, it signs the request and updates the signature value in the respective field of the authorization header (specifically, the "Signature" field) Similarly, in the case of JWT verification, the I/O process verifies the received JWT token from incoming requests using the previously shared JWT secret key.

C. Key Management

After startup, the serverless runtime listens for incoming requests, which can come either through the I/O process or directly from end users—if an application does not contain or handle sensitive data, its communication does not have to involve the I/O module. Once a function is called, an instance of the pre-compiled Wasm code with its own memory and global variables is created to handle the incoming request in an isolated environment.

For sensitive data annotated in a function's code, the data is pre-encrypted during the compilation of the Wasm module. This ensures that LeakLess' encryption keys are not exposed in the runtime's process memory. Encryption keys are generated by the compiler and are stored in a separate file on disk, along with their associated function identifiers. When modules are parsed during the runtime's initialization, only the encrypted data is transferred in memory (and not the key), while the I/O module reads the keys directly from the file.

For sensitive data defined in configuration files, the data is encrypted during the initialization of the serverless runtime, as the configuration file must be loaded first. Any annotated secrets specified by the configuration are encrypted using a newly created AES-128 key, associated with the module's identifier. If the `leakless_operation` attribute is set for a secret, the secret and the type of cryptographic operation that needs to be performed on it are shared with the I/O process. All plaintext sensitive data loaded in memory before encryption, along with the keys used for encryption, are immediately erased from memory and are evicted from the cache. This step occurs *before* any serverless function is instantiated in memory, preventing potentially malicious functions from accessing them.

When generating and assigning encryption keys to serverless applications, LeakLess uses shared keys for functions within the same application, but unique keys for each application. By doing so, functions within the same application can access shared sensitive data, but a malicious tenant cannot decipher others' encrypted data by sharing the same encryption key. If the same key were used across all serverless applications, malicious tenants could gain access to sensitive data in its plaintext form after leaking another tenant's encrypted secret and sending it through the I/O process to an attacker-controlled external service. In this scenario, since the leaked encrypted secret would contain all the required data marshalling prefixes and suffixes, the I/O process would decrypt it prior to sending it to the external service. By using unique keys for each serverless application (and tenant), this attack is prevented.

D. Protecting against Cross-Process Attacks

The I/O module serves as the trusted interface of LeakLess, and mediates all incoming and outgoing messages containing sensitive data. Communication with the main serverless process can be performed through any IPC (inter-process communication) mechanism, but in our implementation with have opted for the use of a network socket (despite the fact that in the most typical deployments both processes will run on the same host). First, this allows for increased flexibility and scalability, in case multiple instances of the serverless runtime (or the I/O module itself) are required for heavy workloads. More importantly, having the ability to transparently run the I/O module on a separate virtual machine or physical host offers increased protection against transient execution attacks that can cross the boundaries of the originating process and leak data from other processes [26]–[30], the OS kernel [39]–[42], or even SGX enclaves [31].

For example, researchers have demonstrated how RIDL [28] (a microarchitectural data sampling attack) can be used by an attacker-controlled renderer process to read arbitrary data from the main web browser process [67]. A similar attack could be mounted by a function to leak data from the I/O module when both run on the same host. To protect against this stronger threat model, as shown in Figure 1, the I/O process can transparently run on a different VM or physical host, without the need for any modifications besides a one-time configuration change.

V. Implementation

WebAssembly and JavaScript are the two main languages recent serverless platforms are based on to isolate functions. After careful consideration of all available open-source serverless platforms at the time, we opted to implement our LeakLess prototype on top of the Spin [44] framework, which is used by Fermyon Cloud. To provide isolation among different tenants, Spin relies on Wasmtime [49] for the concurrent execution of multiple Wasm functions written in various programming languages, such as C++, Rust, and Go.

Besides the core Wasm runtime, serverless platforms based on WebAssembly require additional components, such as socket API support for HTTP networking. Most Wasm runtimes rely on WASI (WebAssembly System Interface) for access to operating system functions, which is under active development [68]. Although the latest version of WASI (Preview 2) supports primitive HTTP communication through `wasi-http` [69], this is not intended to provide all the functionality required by a full-featured cloud execution environment. To address these shortcomings, Spin complements WASI with additional extensions and middleware, such as full HTTP networking support and key–value storage [70].

We developed our prototype on top of Spin v1.1.0, which relies on Wasmtime v7.0.0 and WASI Preview 1. Spin and Wasmtime are written in safe Rust, and we did not use any unsafe code in our modifications to these platforms. We also implemented the I/O module solely in safe Rust, to minimize the risk of memory safety vulnerabilities. Although LeakLess currently supports source code annotations only for Rust and Go, and the String type for holding sensitive data, the overall approach can easily be extended to other languages and data types. Furthermore, LeakLess' support of language-agnostic annotation (Section IV-A1) ensures that all languages supported by Spin can also be accommodated by LeakLess.

A. Annotation and Compilation

1) Annotation in Source Code: We have implemented sensitive data annotation at the source code level for both Rust and Go. For Rust, we rely on Rust's procedural macros, which run during compilation and enable the creation of custom attributes that are attached to items for code manipulation. Our macro identifies and encrypts sensitive strings during compilation. For Go, we rely on the `go:generate` directive, which is used primarily for parsing and modifying code before compilation. In both cases, the mechanism encrypts, encodes, and marshals the data using a unique key that it generates per application, shared across all its Wasm modules.

2) Annotation in Configuration File: In the Spin framework, a variable can be defined in the configuration file, namely `spin.toml`. We leverage this feature to implement language-agnostic annotations at the configuration level. We extended the *Variable* structure within the *manifest* crate, which manages application configurations for the Spin runtime, by adding the two extra `leakless_secret` and `leakless_operation` attributes. This enhancement allows the serverless runtime to recognize and process LeakLess-supported sensitive data and the required operations during runtime for functions written in Spin-supported languages. Therefore, LeakLess can protect functions in any language supported by the Spin framework.

3) Base64 Encoding: Another consideration is that after encryption, the sensitive data will be changed from string to raw binary, and thus it must be converted to Base64 encoding so that it can be transmitted to the I/O module as part of ASCII protocols (e.g., HTTP). Finally, as mentioned in Section IV-A, LeakLess marshals the protected data with a randomly generated prefix and suffix. This approach ensures that when the I/O module receives requests containing encrypted data, it can easily separate the encrypted value from the rest of the stream, then decrypt and replace it. For secrets provided by clients, the client code can follow a similar process.

B. Initialization and Operation

The initialization of serverless functions within the runtime process (Figure 1) is implemented on top of Spin's *HTTP trigger* [70], a web server that listens for HTTP requests from the I/O module and routes them to an *executor*, which instantiates the appropriate Wasm function. In this setup, the manifest plays a crucial role: it not only assigns a unique ID to each Wasm module, but also establishes a mapping between these module IDs and their corresponding HTTP paths (URIs). This approach of mapping module IDs to HTTP paths is a common feature in serverless platforms, facilitating the routing of requests to specific functions or modules based on URL paths. Spin employs a similar mechanism, enabling the Spin trigger to accurately determine which Wasm module should be executed based on the incoming request path. This is particularly useful for requests containing secret data. In such cases, the I/O module first receives the incoming request, and then extracts the encryption key based on the URL path and its associated module ID. Finally, the I/O module encrypts the data before it is forwarded to the serverless runtime.

We also modified Spin's *outbound-http* crate to not only redirect outgoing requests containing sensitive (encrypted) data to the I/O process, but also to include an additional header with the module ID of the module that initiated the outbound request. This header enables the I/O process to easily identify the specific encryption key associated with the module to decrypt of the data.

For language-agnostic annotation, we modified the *loader* crate, which converts the local `spin.toml` file into a configuration executable for the Spin runtime environment. Consequently, when the configuration file is loaded, LeakLess-supported sensitive data is identified and encrypted. Furthermore, if the `leakless_operation` attribute is set for a secret, the serverless function ID, along with the secret's value, are shared with the I/O module via a file.

A unique `cryptographic key` is generated for each serverless application and is used for all functions of the same application. Each cryptographic key along with its corresponding Wasm `module ID` are stored into a file which is then transmitted to the I/O process through a secure connection (to support deployments in which the I/O process runs on a separate host). The keys are then securely erased from the disk, memory, and cache.

The I/O module is built on top of the *hyper* crate, a fast and safe HTTP implementation for Rust that provides both client and server APIs. The I/O module maintains a pool of sockets to the serverless runtime, through which incoming requests from end users are forwarded to the appropriate function, and outgoing requests from functions are forwarded to backend services. In both cases, HTTP messages are inspected for the presence of any sensitive (marshaled) data, which are encrypted or decrypted accordingly. Also, if a serverless function is registered for any `leakless_operation`, the I/O module performs the requested operation on behalf of the serverless function using the shared secret key.

The type of data marshaling we use in LeakLess, which involves utilizing a prefix and suffix, is protocol-agnostic. This approach allows the I/O process to transparently support a wide range of formats and protocols, including XML, JSON, and HTML. While using only a prefix is technically feasible for extracting encrypted or secret data from requests, this method would require the I/O to be customized for each format or protocol. This customization is necessary to identify the length of the data, as different protocols use various separators in their request bodies to separate parameters.

VI. Experimental Evaluation

A. Compatibility Assessment

We performed a thorough analysis of available serverless applications to determine whether they contain sensitive data, and assess if LeakLess can be used to protect them. A "serverless application" is a collection of related functions that interoperate to provide the intended functionality. We found that LeakLess is applicable on the vast majority of real-world serverless functions that handle various types of sensitive data.

1) Data Collection: We collected and analyzed publicly available serverless applications from five sources: the Wonderless dataset [71], the Serverless Framework [72], Fastly [73], Cloudflare [74], [75], and Spin [44], [76]. Wonderless [71] is the largest among the five, and has been used by previous works [46], [77]. The dataset has been collected by crawling GitHub projects and identifying those that use the open-source Serverless Framework [78], which allows developers to deploy applications to various cloud providers. Wonderless uses the presence of the Serverless Framework's default YAML configuration file in a given repository to determine whether the repository uses the framework. We extracted a total of 551 unique applications using the crawler of Wonderless.

We observed that the Wonderless dataset excludes repositories that contain certain keywords (e.g., "example"), and thus it does not include many applications that are provided as examples by the Serverless Framework [78]. Given that these include diverse use cases from various serverless platforms, they serve as a valuable resource for assessing further the compatibility of LeakLess. We thus included 190 additional applications from this repository. Finally, we observed that Wonderless does not adequately cover recent platforms, including Cloudflare [3], Fastly [6], and Spin [44], so we included additional applications from their respective repositories.

Due to the large number of applications, we conducted a comprehensive semi-automated analysis to ascertain the presence and handling of sensitive data in the collected applications. This process began with the automated evaluation of each serverless function's configuration file, focusing on environment variables formatted as key–value pairs [79]. This is a common and recommended practice for passing external values, instead of hard-coding them in the code. Subsequently, we developed a script to systematically extract all variables (defined and assigned), along with any hardcoded strings from the source code. The final step involved a manual review of each extracted variable (from configuration and source code files) to determine its sensitivity and compatibility with LeakLess. The types of sensitive data we identified are similar to those reported by prior studies [45], [46] and those outlined by the platforms themselves [47], and include passwords, secret keys, API tokens, database passwords, and cryptographic keys.

2) Compatibility Analysis: We categorize applications into three groups: 1) *Completely Supported*, in which all sensitive data are fully protected; 2) *Partially Supported*, for those containing both supported and unsupported sensitive data; and 3) *Unsupported*, for those containing sensitive data that cannot be protected. Table I summarizes the number of applications per category for the five datasets. Fewer than half of the applications (449 out of 1,074) contain sensitive data, and from those, LeakLess fully supports 91% of them (407 out of 449). This result underscores the direct applicability of LeakLess on the vast majority of real-world applications. LeakLess partially supports 7% of the applications (33 out of 449), and is incompatible with just nine of them (2%). The 33 partially compatible applications contain a total of 125 sensitive data objects, 70% of which can be successfully protected.

Table II presents a detailed breakdown of the various types of sensitive data *objects* identified across all applications. More than half of the objects (527 out of 966) correspond to authentication secrets used for external APIs or for requests

TABLE I: Sensitive data in serverless applications. LeakLess fully supports 91% (407/449) of those containing sensitive data.

Data Set	**Number of Applications/ Applications with Sensitive Data**	**Fully Supported Applications**	**Partially Supported Applications**	**Unsupported Applications**
Wonderless Dataset [71]	551/299	273	25	1
Serverless Framework [72]	190/65	64	1	0
Fastly [73]	129/11	7	0	4
Cloudflare [74], [75]	145/62	52	7	3
Spin [44], [76]	59/12	11	0	1
Total	1,074/449	407	33	9

TABLE II: Categorization of the different types of sensitive data objects found in serverless applications.

Data Set	**LeakLess Supported Sensitive Data Types**						**Unsupported Sensitive Data Types**		
	Database Password	Database Name	Authentication Secret	Password	JWT Signing or Verification Key	Request Signing Key	Other Crypto Key	Modified Auth. Secret	Modified Password
Wonderless Dataset [71]	41	36	367	27	23	185	18	1	12
Serverless Framework [72]	5	5	60	0	0	37	1	0	0
Fastly [73]	0	0	7	0	0	4	4	0	1
Cloudflare [74], [75]	1	1	88	2	3	3	12	3	0
Spin [44], [76]	5	5	5	2	0	0	1	1	0
Total	52	47	527	31	26	229	36	5	13

within serverless applications (fully supported by LeakLess). We identified five instances of authentication secrets that are generated or modified by the function at run time ("Modified Auth. Secret" in Table II), and thus cannot be protected by LeakLess. Overall, LeakLess supports 94% of the identified sensitive data objects (912 out of 966). The other two types of incompatible data include passwords that are either generated by the function itself or are hashed (column "Modified Password"), and keys used for custom encryption functions, hash computation, and message signing (column "Other Crypto Key"). As we discuss in Section VII, this limitation can be addressed either by outsourcing these operations to the I/O module, similarly to our current support of JWT signing and verification, or by employing register-based encryption [34]. LeakLess fully supports all the remaining data types, including database-related information (passwords, database names), generic user and service passwords, and JWT tokens—the latter through the cooperation of the I/O module for handling signing and verification.

Applications may contain one or more data objects of the same or different types (2.2 objects per application, on average). Figure 2 shows the cumulative distribution of the number of sensitive data objects per application. Nearly 80% of the applications contain just one or two sensitive data objects, while most applications have fewer than six. Considering that only 42% of the applications (449 out of 1,074) handle sensitive data, and that the vast majority of them contain only a handful of data objects, the use of selective data protection is justified, as opposed to securing all data objects of an application.

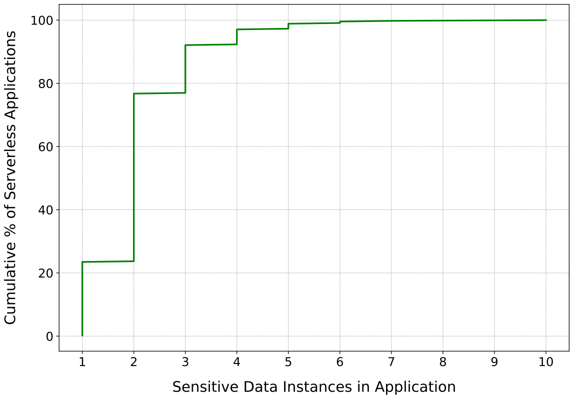


Fig. 2: Cumulative distribution of the number of sensitive data objects per application, as a percentage of all applications containing sensitive data.

3) Real-world Serverless Applications: As each serverless platform has its own set of APIs [80], deploying its applications over the Spin framework requires manual porting effort. Considering the large number of serverless applications that contain sensitive data, it is neither feasible nor necessary to migrate all of them. Since the performance of LeakLess is only influenced by the type of sensitive data, and any necessary LeakLess-supported operations within the applications, we instead identified widely-used representative examples from our datasets, with each example corresponding to a different type of sensitive data and operation. To perform an unbiased selection, we defined an *importance index*, calculated based on the average of several publicly available metrics of a GitHub repository: Stars, Forks, Contributors, Issues, Watchers and Pull Requests. These metrics reflect an application's popularity, activity, and community engagement.

As shown in Table II, LeakLess supports the protection of a diverse range of sensitive data. We used the importance index to identify and extract the most widely used applications from each group of applications that use a given type of sensitive data. This led to the selection of six serverless applications from our entire dataset, detailed in Table III, which we ported to Spin. Specifically, we ported only the functions of each application that contain or handle sensitive data. To replicate external service dependencies, we set up a local version of the respective web services by building a web application with Flask (a Python framework), and then hosting it on our local network using Gunicorn as the web server.

We provide a brief description of each application, focusing on the data protected by LeakLess. applications perform the following six operations respectively: (1) *Authentication Using Stored Tokens* [81]: This example demonstrates the most common scenario in which a function fetches information from an external service using *internal* static sensitive data, i.e., a secret token used for authentication to external APIs. (2) *Authenticating Users at the Edge* [82]: This example is representative of applications that receive *external* sensitive data as a pre-shared secret. This particular function authenticates a user by comparing the received secret with *internal* static sensitive data, i.e., the previously stored version of the secret, before proceeding to provide the requested service. This example highlights the capability of LeakLess to compare two encrypted secrets. (3) *Using Stored Passwords* [83]: Sometimes, secrets such as passwords are stored in the request body and are used for authorization. In this example, a password, stored in the function's source code is used to upload data to an external service. (4) *Signing JWT Keys* [84]: In this example, the function has to generate and *sign* a JWT token that is attached to an outgoing request. When the external service receives the request, it can be sure that it has been legitimately created. LeakLess supports this operation by outsourcing the signing of the JWT token to the I/O module. (5) *Signing Requests* [85]: Most popular cloud-based storage services [58], [86], [87] require two forms of authentication for each request: a secret access token, and a signature generated with a secret key. As in the previous example, the signing operation is outsourced to the I/O module. This particular function involves uploading a file to AWS S3 [58], which verifies the request signature based on the AWS Signature Version 4 format. We replicated the S3 service with the Python web server. (6) *Transmitting User-Provided Secrets* [88]: This example is indicative of cases that involve the transparent flow of sensitive data in and out of a function, as discussed in Section IV-A.

For sensitive data hard-coded directly in the source code, we used the source-level annotation approach (e.g., as in Listing 1 and Listing 2). For sensitive data defined in configuration files, we annotated the respective variables with the LeakLess attribute (e.g., as in Listing 3). When secrets are involved in computation, such as functions responsible for signing requests or JWT tokens, only minimal code adjustments are required. Instead of directly signing the outgoing request or JWT token, the functions were modified to send an empty value as the signature. This change was specifically made in the line of code responsible for signature calculation.

Our current implementation supports only the HTTP protocol, and we were thus unable to evaluate a few database-related applications that use different protocols. However, the forms of authentication used in those scenarios are the same as the ones used in the above examples (e.g., stored credentials, similarly to example (3) above). Therefore, we anticipate that the performance overhead will be comparable.

B. Performance Evaluation

1) Experimental Environment: To evaluate the runtime overhead of LeakLess, we applied it on the ported serverless applications to protect their sensitive data, and tested them using different workloads. The two main sources of overhead are the traffic indirection due to the I/O module, and the additional computation due to the cryptographic operations for encrypting and decrypting the protected data. To discern these two factors, we used a variety of workloads to measure the throughput and latency of a given application for three different configurations: i) vanilla Spin (*"Original"*); ii) with the I/O module but without encryption (*"I/O Only"*); and iii) with the I/O module and cryptographic operations, i.e., full LeakLess protection (*"I/O + Encryption"*).

We used the well-known benchmarking tool `wrk` v4.2.0 for workload generation. To reduce the chance of measurement errors, we repeated each experiment 10 times and report the average of each measurement. In each run the client spawns 30 threads that send a total of 1,000 concurrent requests per second for a period of 10 minutes. To make the evaluation fair and reduce the possibility of experimental error, we restart the whole platform before repeating each experiment.

We ran our experiments on a server equipped with an Intel Xeon E3-1240 CPU and 32GB of RAM, running Ubuntu 20.04.5 and kernel v5.4.0-128. We evaluated the above three configurations in the two main deployment scenarios supported by LeakLess: running the I/O module on the same host as Spin (*local*), and running it on a different physical machine (*remote*). The client machine used for workload generation is equipped with an Intel Core i7-6700 CPU and 32GB of RAM, running Ubuntu 22.04.1 and Linux kernel v5.15.0-56. We also used two additional servers for hosting the Python web server simulating external services, and the I/O module in the remote scenario, both running Ubuntu 22.04: one equipped with an Intel Core i7-7700 CPU and 32GB of RAM, and the other with an Intel Core i7-4790 CPU and 32GB of RAM. All machines are interconnected through a 1 Gbit/s network switch.

2) Performance Overhead: We evaluate the performance impact of LeakLess when protecting the sensitive data of each application. The size of the protected data ranges from 20 to 46 bytes, depending on the particular application. The size of user requests and responses from the web server are smaller than 1KB, in accordance to the real external services used by each application, except for the "Signing Requests" application, which involves uploading a 50KB file to an external service.

The results of our experiments are reported in Table III. We measured throughput degradation (requests per second) and latency increase (ms) for the local and remote scenarios. Numbers in parentheses correspond to the percentage of increase compared to the baseline (vanilla Spin). Overall, LeakLess introduces a latency increase of up to 3.4% and throughput decrease of up to 2.8% for the remote scenario,

TABLE III: Throughput reduction and latency increase for six real-world serverless applications, when the I/O module runs on a different host (Remote) and on the same host with Spin (Local). Values in parentheses indicate the percentage change in throughput or latency compared to the original.

Application	Latency (ms)					Throughput (req/s)				
	Orig.	I/O Only		I/O + Encryption		Orig.	I/O Only		I/O + Encryption	
		Remote	Local	Remote	Local		Remote	Local	Remote	Local
Authentication Using Stored Tokens [81]	1,267	1,310 (3.3)	1,356 (7.0)	1,310 (3.3)	1,357 (7.1)	769	747 (2.8)	723 (5.9)	747 (2.8)	723 (5.9)
Authenticating Users at the Edge [82]	1,285	1,290 (0.4)	1,310 (1.9)	1,290 (0.4)	1,320 (2.7)	766	764 (0.0)	749 (2.2)	765 (0.0)	739 (3.5)
Using Stored Passwords [83]	1,267	1,310 (3.4)	1,356 (7.0)	1,310 (3.4)	1,356 (7.0)	769	749 (2.6)	722 (6.1)	751 (2.3)	722 (6.1)
Signing JWT Keys [84]	1,240	1,280 (3.2)	1,350 (8.8)	1,280 (3.2)	1,360 (9.6)	788	766 (2.7)	722 (8.3)	766 (2.7)	721 (8.5)
Signing Requests [85]	1,466	1,470 (0.0)	1,560 (6.4)	1,470 (0.0)	1,574 (7.3)	666	660 (0.0)	632 (5.1)	660 (0.0)	620 (6.9)
Transmitting User-Provided Secrets [88]	1,340	1,360 (1.4)	1,460 (6.9)	1,365 (1.8)	1,470 (9.7)	730	724 (0.8)	671 (8.0)	723 (0.9)	670 (8.2)

and a latency increase of up to 9.7% and throughput decrease of up to 8.5% for the local scenario.

We should note that the results for the remote scenario misrepresent the overall *computational* overhead, as the lower latency increase and throughput reduction obscure the fact the *additional* computation is still expended by the second host. To accurately assess this extra CPU overhead, we used two separate LeakLess instances on two machines, each running its own as well as the other instance's I/O module, and measured the combined latency increase and throughput reduction. As expected, we observed that the overhead per machine is essentially the same as in the above *local* scenario. Therefore, for the remote scenario, the overall computational overhead is more accurately reflected by the local scenario results.

Across the board, comparing the "I/O Only" and "I/O + Encryption" configurations, we observe that the computational overhead due to the extra cryptographic and data marshaling operations is negligible. This is expected due to the relatively small size of the sensitive data across all applications. We explored whether significantly increasing the number and size of sensitive data objects affects overhead through additional stress-testing experiments, presented in Appendix A.

The variation in the overheads for different applications are related to each application's functionalities and the specific LeakLess operations performed by the I/O module. For example, both the first and third applications use a secret for authentication with backend services, placed in the header and in the body, respectively. In I/O-bound applications, where performance is predominantly limited by data transfer and disk access times, the impact of LeakLess' additional processing overhead becomes less significant. In the "Signing Requests" application, for example, the primary operation involves reading and transferring a file, which are inherently I/O-intensive. Similarly, the "Authenticating Users at the Edge" application stores data in a key–value store backed by an SQLite database, another I/O-centric task. In such scenarios, the bulk of execution time is consumed by I/O operations, making the additional processing time introduced by LeakLess negligible. Furthermore, our proxy module is implemented using Rust's hyper crate [89], acclaimed for its fast and safe HTTP implementation. We also use asynchronous programming for implementing LeakLess-related tasks within the I/O module, which enhances throughput and responsiveness. Rust's ownership model adds to this efficiency, ensuring memory safety without a garbage collector, which is particularly beneficial in concurrent environments.

Conversely, when the I/O module and Spin operate on the same host, we observe an increased overhead. This difference can be attributed to the competing resource requirements of the I/O module and the main runtime process. In the remote scenario, each process operates on its own dedicated hardware, including CPU, memory, and network bandwidth, without any interference. On the other hand, in the local scenario, both the I/O module and the Spin framework contend for the same resources on the same host, leading to increased overhead. It is thus preferable to host the I/O module on a separate host, as this offers increased protection against transient execution attacks, while minimizing performance overhead.

3) Functionality and Scalability Evaluation: To verify that LeakLess does not affect the correct operation of the tested applications, we use a script to call each modified function and log the output. As serverless functions are designed for single-purpose computations, we assess whether the response adheres to the expected outcome, such as successful data handling and correct behavior under various operational conditions. For example, in the "Signing Requests" application, when a file is successfully uploaded to the web server, the function returns a status code indicating success. Notably, since 72% of the data handled by LeakLess is immutable, its impact on application functionality is consistent, and thus easy to verify that it does not break any functionality. We also tested several cases involving mutable secrets (e.g., request signing, JWT tokens), where the algorithms are consistent across applications. Our tests were realistic, including interactions with real external services (e.g., fetching images from Amazon S3 buckets).

To evaluate scalability using more than one I/O modules, we successfully evaluated different scenarios involving up to three I/O modules using the same runtime. In all cases, we observed stable throughput and no significant deviations compared to the single I/O module scenario.

4) Comparison with Previous Works: LeakLess protects against all types of leakage, whereas previous approaches protect only against certain Spectre variants. Swivel [32] explicitly does not protect against RIDL and Meltdown-style attacks. As the authors of DyPrIs [22] acknowledge, as an anomaly detection method, it may suffer from false positives/negatives. Given these qualitative differences, a direct quantitative comparison would not be meaningful. Despite the qualitative differences we still explored whether such a comparison is possible, as the source code of Swivel is publicly available (the code of DyPrIs is not). Unfortunately, we could not perform a direct quantitative comparison with LeakLess due to multiple compatibility issues with the underlying frameworks. There are several reasons for this incompatibility. First, Swivel was implemented on the deprecated Lucet runtime [90], which is incompatible with any open-source serverless frameworks. Second, Swivel was implemented on an earlier version of Wasmtime (v0.15.0) [91], which is not compatible with any version of the Spin framework.

The performance impact of Swivel was evaluated using the Rocket web server, which has the capability of hosting web services compiled to WebAssembly modules on top of a Wasm runtime. However, by design, the real-world serverless applications we collected are not compatible with this configuration. The main reason is that the Lucet runtime, on which Swivel relies, uses WASI Preview 1 (WebAssembly System Interface) [51] to access operating system resources, but this WASI version does not support HTTP communication or any other higher-level APIs used by serverless functions.

As discussed in Section V, serverless platforms such as Spin expose HTTP and other APIs to functions, enabling them to perform typical web application operations. Instead, Rocket essentially uses Wasm modules in a `cgi-bin` fashion, offering only a `stdin`/`stdout` interface, i.e., not providing a true serverless platform. Consequently, it is not possible to run existing serverless applications involving HTTP communication directly on the Rocket web server—the closest functionality is to forward the raw HTTP request to the relevant module's standard input. Indeed, Swivel was evaluated using modules written in C (and compiled to Wasm) that perform general-purpose operations, such as expanding HTML templates or converting XML input to JSON output.

The reported throughput reduction for Swivel using these modules ranges between 28.4% and 33.7%, which is an order of magnitude higher than LeakLess. We should note that according to our experiments, the overhead of LeakLess for these same applications would have been negligible, as only any potential sensitive objects within HTML or XML files would need to be modified, and not the whole file.

More importantly, however, the performance impact characteristics of Swivel and LeakLess in real-world deployments are completely different. At a high level, Swivel prevents a malicious Wasm module (i.e., serverless function) from accessing the data of *other modules*. Deploying Swivel thus necessitates the assumption that *any* loaded module can potentially be malicious. This means that Swivel has to be applied to *all* the loaded modules from *all* tenants that run as part of the same runtime—affecting the performance of all of them. In contrast, LeakLess *selectively* protects a benign function from any other potentially malicious function, and thus needs to be applied *only* to functions that handle sensitive data (at the discretion of the developer), without affecting the performance of other functions and the platform as a whole.

C. Security Evaluation

1) Transient Execution Attacks: In the default Spin setup, functions from multiple tenants run in the same process, and thus share the same virtual memory address space. Each function runs in its own *sandbox* with its own allocated memory and set of capabilities, which are limited by the WebAssembly runtime. However, this level of isolation does not protect against transient execution attacks [19]. Although such attacks typically depend on high-precision timers, limiting the use of precise timing mechanisms is not an effective mitigation, as attackers have developed alternative methods. These include coarse-grained or remote timers [22], [92]–[94], counting threads to approximate cycle counters [95], [96], leveraging cache traces with performance counters [97], synchronization-based storage channels [98], and exploiting architectural dependencies [59].

As a concrete example, Schwarzl et al. [22] successfully executed a Spectre attack against the Cloudflare Workers FaaS platform, which is based on a modified JavaScript V8 sandbox that disables all known timers and primitives (e.g., the `rdtsc` CPU instruction). Despite this mitigation, the authors leveraged amplification techniques and a remote timing server to leak secret data from other workers using Spectre gadgets. This example underscores that disabling timers is insufficient to thwart all speculative execution attacks [99]. Similarly, the Wasmtime runtime [49] only implements basic Spectre mitigations, such as bounds checking for `call_indirect` instructions. These and other mitigation techniques, including retpoline [100] and user pointer sanitization [101], are not effective against the latest attack variants such as Retbleed [41].

Based on the above, we consider the attack scenario where the adversary uses a memory disclosure vulnerability or a transient execution attack to leak secrets from other tenants. To carry out transient execution attacks, given that the low-level instructions used in these exploits [20], [102] (e.g., `rdtsc`) are not available to Wasm code, we assume that the attacker will use other techniques (such as a remote server) to construct precise timers [103], or will perform the attacks without relying on an architectural timer at all [59]. The attacker runs a malicious serverless function containing a self-crafted Spectre gadget that performs a Spectre attack on its own process to leak secrets from a victim function—there is no need to discover an existing Spectre gadget within the victim function, since they are both part of the same process.

We recreated this attack scenario to verify that sensitive data is never exposed in the runtime's process memory in unencrypted form. To address every scenario in which an attacker could extract data from another tenant, we implemented a patch for Spin that causes all memory of a Wasm function to be exposed when triggered by an HTTP request. The patch then analyzes the exposed memory to confirm that no secrets are present in unencrypted form. This approach ensures that across all real-world applications (Section VI-B), the secrets are never exposed in memory.

2) Data Leakage Vulnerabilities: Besides transient execution attacks, bugs in the language runtime itself uphold the threat of memory leakage attacks. Several flaws in the implementation of Wasmtime, which is used by Spin to manage these language-based sandboxes, have been discovered [13]–[15]. These flaws enable a malicious function to access the memory of other functions and leak other tenants' sensitive data. Notably, such vulnerabilities are not unique to Wasmtime, as similar issues have been found in V8 isolates [11], [12], indicating a broader challenge in ensuring robust language-level sandboxing across different platforms. As we showed in the previous experiment, sensitive data is never exposed in decrypted form in the function's memory.

To also cover the case in which an attacker might attempt to read data directly from the runtime's memory, we developed a custom program that dumps the memory of the main process after the initialization step is completed. This is a plausible scenario, as exemplified by a recent bug [15] in Wasmtime's Cranelift code generator, which enables Wasm modules to access memory up to 34GB away from their base address. The program uses the `gcore` tool to attach to the runtime process and dump its memory. We then scanned these memory dumps for the presence of the plaintext version of the protected data, as well as the cryptographic keys used by LeakLess for encryption in the runtime process, to verify that they have been erased from memory before allowing any function to be executed. We confirmed that both the protected data and the keys were never found in unencrypted form.

3) Confused Deputy Attacks: To minimize the risk of data collisions, we rely on industry-standard data marshaling requirements, using 128-bit random prefixes and suffixes. Given that the attacker has access as a tenant of the serverless framework, these prefixes and suffixes are not considered as secret. However, to mitigate the potential for confused deputy attacks, in which an attacker could misuse the I/O module to decrypt other tenants' secrets, we use distinct cryptographic keys for each application, shared across all its Wasm modules. Assume an attacker manages to leak an encrypted secret from memory. In case of an internal secret, the attacker could integrate the leaked secret into their own Leakless-protected function, which transmits the secret to a server under their control. Although the I/O module will decrypt the secret, it will do so using a different key (the key generated for the attacker's application), resulting in incorrect data. In case of an external secret, the attacker could treat it as an external input and replay it to the victim function, but in that case the already encrypted secret will just be re-encrypted by the I/O module. Replaying the secret to an attacker-controlled function is again ineffective due to the use of a different key per application.

VII. Limitations and Discussion

Performance Optimizations: Although LeakLess incurs a modest performance overhead even when the I/O module and the serverless runtime run on the same host, further optimization is still possible. Our prototype uses TCP sockets with a connection pool for data transfer between the runtime process and the I/O module. This approach was chosen to transparently support running the I/O module on a separate host (to defend against cross-process attacks). More efficient inter-process communication mechanisms, such as Unix domain sockets or shared memory, can be considered when running the runtime and the I/O module on the same host.

Immutable Data: LeakLess currently supports the protection of sensitive data that remains unchanged during function execution. In addition, it supports commonly found cryptographic operations that involve sensitive data, such as signing and verification, which are often performed by serverless functions when communicating with external services. However, as discussed in Section VI-A, there are still a few cases of sensitive data that are not currently supported by LeakLess, mostly related to keys used for custom encryption functions, hash computation, and message signing (54 out of 966 identified sensitive data objects).

This is a limitation of our implementation, which can be addressed with additional engineering effort. One approach is to outsource these additional cryptographic operations to the I/O module, similarly to the current support for JWT operations. As shown in Table III, this would not impact overall performance significantly. A more generic solution would be to implement support for performing arbitrary computation on encrypted data. This can be achieved using a register-based encryption scheme [33], [34], which would allow the function itself to safely decrypt the protected data when loaded into CPU registers, carry out the desired computation, and re-encrypt it before writing it in memory. We chose not to pursue this approach due to the significant engineering effort required and the relatively few cases needing this level of support.

Despite this limitation, as demonstrated in Section VI-A, LeakLess is directly applicable on the vast majority of real-world serverless functions that handle sensitive data. One of their common characteristics is that they typically carry out short-lived operations with data provided by external sources that involve secure interaction with external cloud services and APIs [2], [4], [58]. Notably, 70% of the identified secret objects in Table II fall in this category. LeakLess can be applied directly in these scenarios to provide robust protection with minimal developer effort, given that sensitive data such as authentication tokens, credit card information, session IDs, database credentials, and authorization tokens remain immutable for the whole duration of a function's execution.

Compatibility with Existing Platforms: Despite the introduction of a separate I/O module, the overall design of LeakLess remains compatible with existing serverless platforms. In general, the use of proxies to intercept requests is a common technique used by previous works that focus on protecting data in serverless environments [45], [104]. Similarly, Cloudflare Workers [3] enforce outbound restrictions and handle inbound redirection through two proxy services [25]. These *existing* proxies can be easily extended to perform the cryptographic operations on the protected data required by LeakLess. As part of our future work, we plan to integrate LeakLess in the recently released open-source version of CloudFlare's serverless runtime, which now also supports WebAssembly [105].

VIII. Related Work

We discuss various types of defenses that can be used against data-only attacks [106] and transient execution attacks [19], [20], targeting both native and WebAssembly programs. Additionally, we discuss defenses that are specifically tailored to serverless platforms.

Memory Safety: There have been many works which attempt to address vulnerabilities caused by the usage of memory unsafe languages such as C/C++. Techniques like SoftBound [107] maintain bounds information for each pointer, providing spatial safety, but incur high runtime overhead. Software-based defenses such as DataShield [108] enforce memory safety at an object level, protecting against data leakage attacks. Lehmann et al. [109] demonstrate that vulnerabilities with mitigations in C/C++ code can propagate to WebAssembly binaries, highlighting the importance of Memory Safe WebAssembly (MSWasm) [110], which extends Wasm with memory safety abstractions.

Some works have focused on enhancing the security of serverless platforms against data-only attacks. An example is Groundhog [111], which secures serverless systems by returning to a clean state after each function call to remove private data from the process address space. SecWasm [112] implements an information flow control system to guarantee the safe handling of sensitive information in Wasm. In comparison to LeakLess, which safeguards serverless functions against Spectre attacks, these prior works do not consider transient execution attacks as a potential security threat.

Isolation-based Defenses: Various isolation strategies have been explored to protect program data. One method is privilege separation, which aims to reduce the code running with special privileges without affecting program functionality [113], [114]. Chrome and Firefox, for example, employ *site isolation*, which loads every page (browser tab) in its own process [115]–[117]. However, in certain cases, site isolation is still vulnerable to transient execution attacks [118], while cross-process [26]–[30] and user-to-kernel [39]–[42] attacks are also possible.

DyPrIs [22] uses anomaly detection to identify malicious workers in the Cloudflare Workers platform [3] that may be mounting transient execution attacks, which are then migrated into a separate process. Similarly to other anomaly-based approaches, DyPrIs suffers from false positives and negatives, while it still cannot protect against cross-process transient execution attacks [26]–[31], [39].

Hardware-based isolation is another approach, using hardware extensions like Intel SGX and MPK to protect applications from data-only attacks [35], [119]–[123]. However, these techniques often exclude transient execution attacks or lack applicability to serverless cloud computing platforms. TME-Box [124] leverages Intel TME-MK for scalable in-process isolation in cloud computing environments, but similarly does not address transient execution attacks.

One approach to strengthen the isolation guarantees of WebAssembly is to verify the generated code or attempt to detect vulnerabilities [16]–[18]. WaVe [125] is a Wasm runtime system that uses automated verification to ensure memory isolation and proper access restriction to OS resources. WebAssembly has been used as a means to provide isolation by several works [126]–[128]. These works leverage WebAssembly to isolate parts of the code which are deemed as untrusted. However, software-based isolation alone is insufficient to protect against transient execution attacks.

Defenses Against Transient Execution Attacks: We summarize prior efforts to address Spectre-like attacks in different environments. Several methods aim to eliminate the source of data leakage. For instance, CleanupSpec [129] uses an "undo logic" for the cache state to prevent data leakage. Some approaches introduce barriers (e.g., using the `lfence` instruction), to eliminate Spectre gadgets [130], [131]. Retpoline [100] mitigates Spectre by hardening all branch instructions against speculative execution. SpecLFB [132] leverages the Line-Fill-Buffer (LFB), a microarchitectural component of CPUs, to eliminate potential side channels before they are established.

To protect against indirect branch attacks, SpecCFI [133] uses control flow integrity to validate the targets of indirect branches during speculative execution. Half&Half [134] isolates and prevents malicious mistraining of conditional branch predictors (CBP) in modern Intel processors by physically partitioning all CBP structures based on a single bit of the branch address. Most of these methods are not applicable for serverless platforms due to either their significant performance overhead or compatibility issues. An alternative approach is to assume that leakage will happen, and ensure that sensitive data will always remain encrypted in memory [33], [34]. These methods effectively secure internal application data, but are inadequate for sensitive data that need to be transmitted outside the application (i.e., data is decrypted before being sent).

Swivel [32] is a compiler-based technique that uses software-based and hardware-based fault isolation to secure Wasm applications. This additional instrumentation incurs a considerably high overhead, especially for its deterministic versions. Swivel assumes all memory locations contain potentially sensitive data, whereas LeakLess only protects specific confidential information. Wasm-Mutate [135] introduces a method for diversifying WebAssembly binaries to mitigate timing side-channel attacks, but the approach lacks soundness. Narayan et al. [136] propose Hardware-assisted Fault Isolation (HFI) to improve isolation for WebAssembly and native binaries and address shortcomings in software-based systems against transient attacks. Nevertheless, it requires modifications to the hardware, operating system, and runtime.

IX. Conclusion

We presented LeakLess, a software architecture for selective data protection tailored to serverless platforms, which provides future-proof protection against memory disclosure and transient execution attacks. LeakLess introduces a separate I/O module that mediates the handling of sensitive data between the serverless runtime and external entities, ensuring that secrets always remain encrypted in the runtime's memory. The only manual effort required from the side of developers to protect sensitive data is to just annotate the respective variables in the source code of serverless functions, and in any requests sent by clients. We implemented LeakLess by modifying the Spin framework, which uses the Wasmtime runtime to run Wasm modules in the same runtime process. We evaluated the usability and performance of LeakLess with real-world serverless functions. Due to its narrow-scope cryptographic operations, LeakLess incurs only negligible overhead for typical applications, while protecting against both intra-process and cross-process transient execution attacks.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. We also thank Martin Schwarzl and Tapti Palit for their valuable comments and discussions, and the developers of the Spin framework for their helpful guidance. This work was supported by the National Science Foundation (NSF) through award CNS-2104148.

References

[1] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Comput. Surv.*, vol. 54, no. 10s, 2022.

[2] "Amazon Lambda," https://aws.amazon.com/lambda/, 2024.

[3] "Cloudflare Workers," https://workers.cloudflare.com/, 2024.

[4] "Google Cloud Products," https://cloud.google.com/products, 2024.

[5] "Azure Functions," https://azure.microsoft.com/en-us/products/functions/, 2024.

[6] "Fastly Compute@Edge," https://www.fastly.com/products/edge-compute/serverless, 2024.

[7] A. Sahraei, S. Demetriou, A. Sobhgol, H. Zhang, A. Nagaraja, N. Pathak, G. Joshi, and et al, "XFaaS: Hyperscale and low cost serverless functions at Meta," in *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 231–246.

[8] "Deno: Next-generation JavaScript runtime," https://deno.com/, 2024.

[9] "V8," https://v8.dev/, 2024.

[10] "WebAssembly," https://webassembly.org/, 2024.

[11] "Google Chromium V8 Out-of-Bounds Memory Access Vulnerability," https://nvd.nist.gov/vuln/detail/cve-2024-0519, 2024.

[12] "Stable Channel Update for Desktop," https://chromereleases.googleblog.com/2024/01/stable-channel-update-for-desktop_16.html, 2024.

[13] "Memory Access Due to Code Generation Flaw in Cranelift Module," https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-hpqh-2wqx-7qp5, 2021.

[14] "Data leakage Between Instances in the Pooling Allocator," https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-wh6w-3828-g9qf, 2022.

[15] "Guest-controlled out-of-bounds read/write on x86_64," https://github.com/bytecodealliance/wasmtime/security/advisories/GHSA-ff4p-7xrq-q5r8, 2023.

[16] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust but verify: SFI safety for native-compiled Wasm," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[17] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-Safe multilingual software sandboxing using WebAssembly," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[18] "Cargo Fuzz Targets for Wasmtime," https://github.com/bytecodealliance/wasmtime/blob/main/fuzz/README.md, 2024.

[19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security & Privacy (S&P)*, May 2019, pp. 903–101.

[20] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. V. Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 249–266.

[21] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for software Spectre defenses," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[22] M. Schwarzl, P. Borrello, A. Kogler, V. Kenton, T. Schuster, M. Schwarz, and D. Gruss, "Robust and scalable process isolation against Spectre in the cloud," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2022, pp. 167–186.

[23] "Introducing Secrets and Environment Variables to Cloudflare Workers," https://blog.cloudflare.com/workers-secrets-environment/, 2024.

[24] "Announcing Cloudflare Secrets Store," https://blog.cloudflare.com/secrets-store/, 2024.

[25] K. Varda, "Mitigating Spectre and Other Security Threats: The Cloudflare Workers Security Model," https://blog.cloudflare.com/mitigating-spectre-and-other-security-threats-the-cloudflare-workers-security-model/.

[26] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[27] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018, pp. 2109–2122.

[28] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 88–105.

[29] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on Meltdown-resistant CPUs," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 769–784.

[30] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 753–768.

[31] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SGXPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 142–157.

[32] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[33] T. Palit, J. F. Moon, F. Monrose, and M. Polychronakis, "DynPTA: Combining static and dynamic analysis for practical selective data protection," in *Proceedings of the 42nd IEEE Symposium on Security & Privacy (S&P)*, 2021, pp. 1919–1937.

[34] T. Palit, F. Monrose, and M. Polychronakis, "Mitigating data leakage by protecting memory-resident sensitive data," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 598–611.

[35] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel., "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proceedings of the USENIX Security Symposium*, 2019, pp. 1221–1238.

[36] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient in-process isolation for RISC-V and x86," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 1677–1694.

[37] L. Li, H. Yavarzadeh, and D. Tullsen, "Indirector: High-precision branch target injection attacks exploiting the indirect branch predictor," in *Proceedings of the 33rd USENIX Security Symposium*, 2024.

[38] J. Wikner and K. Razavi, "Breaking the barrier: Post-barrier Spectre attacks," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2025.

[39] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[40] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks," in *Proceedings of the 31st USENIX Security Symposium*, 2022, pp. 971–988.

[41] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *Proceedings of the 31st USENIX Security Symposium*, 2022, pp. 3825–3842.

[42] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative dereferencing: Reviving Foreshadow," in *Proceedings of the 25th International Conference on Financial Cryptography and Data Security (FC)*, 2021, pp. 311–330.

[43] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, "InSpectre gadget: Inspecting the residual attack surface of cross-privilege Spectre v2," in *Proceedings of the 33rd USENIX Security Symposium*, 2024.

[44] "Spin," https://developer.fermyon.com/spin/index, 2024.

[45] P. Datta, P. Kumar, T. Morris, M. Grace, A. Rahmati, and A. Bates, "Valve: Securing function workflows on serverless computing platforms," in *Proceedings of The Web Conference (WWW)*, 2020, pp. 939–950.

[46] D. S. Jegan, L. Wang, S. Bhagat, and M. Swift, "Guarding serverless applications with Kalium," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[47] "Annotation traits in wasmCloud." https://wasmcloud.com/docs/hosts/abis/wasmbus/interfaces/traits, 2024.

[48] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM Conference on Programming Language Design and Implementation (PLDI)*, 2017, pp. 185–200.

[49] "Wasmtime," https://wasmtime.dev/, 2024.

[50] "WasmEdge," https://wasmedge.org/, 2024.

[51] "WASI," https://wasi.dev/, 2024.

[52] Gadepalli, P. Kishore, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight Wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 265–279.

[53] Gadepalli, P. Kishore, G. Peach, L. Cherkasova, R. Aitken, and G. Parmer, "Challenges and opportunities for efficient serverless computing at the edge," in *Proceedings of the 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 261–2615.

[54] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, 2019, pp. 225–236.

[55] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Proceedings of the 27th European Symposium on Research in Computer Security (ESORICS)*. Springer, 2019, pp. 279–299.

[56] "Heroku," https://www.heroku.com/, 2024.

[57] "AWS Products," https://aws.amazon.com/products/, 2024.

[58] "Amazon S3," https://aws.amazon.com/s3/, 2024.

[59] R. Zhang, T. Kim, D. Weber, and M. Schwarz, "(M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels," in *Proceedings of the USENIX Security Symposium*, 2023.

[60] H. Xiao and S. Ainsworth, "Hacky racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023, pp. 354–369.

[61] A. Purnal, M. Bognar, F. Piessens, and I. Verbauwhede, "ShowTime: Amplifying arbitrary CPU timing side channels," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2023.

[62] Z. Weissman, T. Tiemann, T. Eisenbarth, and B. Sunar, "Microarchitectural security of AWS Firecracker VMM for serverless cloud platforms," 2023.

[63] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[64] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 957–972.

[65] "Data Annotations for Model Validation," https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-6, 2024.

[66] "Configuration of wrangler.toml," https://developers.cloudflare.com/workers/wrangler/configuration/, 2024.

[67] S. Röttger, "Escaping the Chrome Sandbox with RIDL," https://googleprojectzero.blogspot.com/2020/02/escaping-chrome-sandbox-with-ridl.html.

[68] "WASI Cloud Core Proposal," https://github.com/WebAssembly/wasi-cloud-core, 2024.

[69] "WASI Preview 2," https://github.com/WebAssembly/WASI/tree/main/preview2, 2024.

[70] F. Developer, "The Spin HTTP Trigger," https://developer.fermyon.com/spin/v2/http-trigger, 2024.

[71] N. Eskandani and G. Salvaneschi, "The Wonderless dataset for serverless computing," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, 2021, pp. 565–569.

[72] "Serverless Framework Serverless Application Examples," https://github.com/serverless/examples, 2024.

[73] "Fastly Serverless Application Code Examples," https://developer.fastly.com/solutions/examples/, 2024.

[74] "Cloudflare Workers serverless application examples," https://developers.cloudflare.com/workers/examples/, 2024.

[75] "Cloudflare Workers Supported Packages Examples," https://airtable.com/embed/shrTR0QCusxZoCgiJ/tbloKKErinTfrIHsB, 2024.

[76] "Spin GitHub Repository," https://github.com/fermyon/spin, 2024.

[77] X. Liu, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "FaaSLight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 5, jul 2023.

[78] "Serverless Framework," https://www.serverless.com/, 2024.

[79] "Serverless Platform Parameters," https://www.serverless.com/framework/docs/guides/parameters, 2024.

[80] T. Lienard, "V8 Isolates Are Taking Over the World," https://dev.to/tomlienard/v8-isolates-are-taking-over-the-world-3h4m, 2022.

[81] "CDN edge endpoint," https://github.com/kriasoft/react-starter-kit/tree/b882d5a759b3d344fe390205e0db68387c9057de/edge, 2024.

[82] "Open Charge Map using Cloudflare API Router," https://github.com/openchargemap/ocm-system/tree/master/API/OCM.Net/OCM.API.Worker/cloudflare/api-router, 2024.

[83] "Lambda function for extracting text from a variety of file types," https://github.com/Enterprise-CMCS/cmcs-eregulations/tree/8cc15ef111c2c909f849b5dd02ed54008dbc8ca8/solution/text-extractor, 2024.

[84] "Scheduler Package of the Framework for adding A/B testing to education applications," https://github.com/CarnegieLearningWeb/UpGrade/tree/dev/backend/packages/Scheduler, 2024.

[85] "Sign Request Example: Splice Large Media Files and Upload the Spliced Media Files to Cloud Storage." https://github.com/tencentyun/serverless-demo/tree/9a2bc6274f9970ba05e5beb5f41b109dd885e0ee/Python3.6-MediaConcat/src, 2024.

[86] "Azure Archive Storage," https://azure.microsoft.com/en-us/products/storage/#overview, 2024.

[87] "Object Storage for Companies of All Sizes," https://cloud.google.com/storage/?hl=en, 2024.

[88] "Notion as CMS with easy API access," https://github.com/splitbee/notion-api-worker/, 2024.

[89] "Hyper Crate," https://hyper.rs/, 2024.

[90] "Lucet," https://github.com/bytecodealliance/lucet, 2024.

[91] "Wasmtime Spectre," https://github.com/PLSysSec/wasmtime-spectre, 2024.

[92] L. Hetterich and M. Schwarz, "Branch different - Spectre attacks on Apple silicon," in *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2022, pp. 116–135.

[93] "A Spectre proof-of-concept for a Spectre-proof web," https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021.

[94] "Spectre JavaScript PoCs," https://leaky.page/, 2021.

[95] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 549–564.

[96] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, 2017, pp. 247–267.

[97] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 639–656.

[98] J. Yu, A. Dutta, T. Jaeger, D. Kohlbrenner, and C. W. Fletcher, "Synchronization storage channels (S2C): Timer-less cache side-channel attacks on the Apple M1 via hardware synchronization instructions," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[99] J. Zhang, C. Chen, J. Cui, and K. Li, "Timing side-channel attacks and countermeasures in CPU microarchitectures," in *ACM Computing Surveys 56*, 2024, pp. 1–40.

[100] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886.

[101] "Spectre Side Channels," https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/spectre.rst.

[102] "SafeSide," https://github.com/google/safeside, 2024.

[103] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand pwning unit: Accelerating microarchitectural attacks with the GPU," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[104] A. Sankaran, P. Datta, and A. Bates, "Workflow integration alleviates identity and access management in serverless computing," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020, pp. 496–509.

[105] "Workerd, Cloudflare's JavaScript/Wasm Runtime," https://github.com/cloudflare/workerd, 2024.

[106] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[107] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," in *Proceedings of the 30th ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 245–258.

[108] S. A. Carr and M. Payer, "DataShield: Configurable data confidentiality and integrity," in *Proceedings of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*, 2017, pp. 193–204.

[109] D. Lehmann, J. Kinder, and M. Pradel, "Old is new again: Binary security of WebAssembly," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 217–234.

[110] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan, "MSWasm: Soundly enforcing memory-safe execution of unsafe code," *Proc. ACM Program. Lang.*, vol. 7, 2023.

[111] M. Alzayat, J. Mace, P. Druschel, and D. Garg, "Groundhog: Efficient request isolation in FaaS," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, 2023, pp. 398–415.

[112] I. Bastys, M. Algehed, A. S. osten, and A. Sabelfeld, "SecWasm: Information flow control for WebAssembly," in *Proceedings of the International Static Analysis Symposium*, 2022, pp. 74–103.

[113] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for Intel SGX," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017, pp. 285–298.

[114] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative privilege separation," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019, pp. 1023–1040.

[115] C. Reis, A. Moshchuk, and N. Oskov, "Site isolation: Process separation for web sites within the browser," in *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1661–1678.

[116] "Site isolation," https://www.chromium.org/Home/chromium-security/site-isolation/, 2024.

[117] "Security/Sandbox - MozillaWiki," https://wiki.mozilla.org/Security/Sandbox, 2024.

[118] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook.js: Attacking Chrome strict site isolation via speculative execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 699–715.

[119] J. Gu, H. Li, W. Li, Y. Xia, and H. Chen, "EPK: Scalable and efficient memory protection keys," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022, pp. 609–624.

[120] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 563–577.

[121] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu., "Faastlane: Accelerating function-as-a-service workflows," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2021, pp. 805–820.

[122] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-FaaS: Trustworthy and accountable function-as-a-service using Intel SGX," in *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, 2019, pp. 185–199.

[123] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer, "Clemmys: Towards secure remote execution in FaaS," in *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, 2019.

[124] U. Martin, L. Lamster, D. Schrammel, M. Schwarzl, and S. Mangard, "TME-Box: Scalable in-process isolation through Intel TME-MK memory encryption," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2025.

[125] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown, "WaVe: a verifiably secure WebAssembly sandboxing runtime," in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2023, pp. 2940–2955.

[126] S. Shillaker and P. Pietzuc, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2020, pp. 419–433.

[127] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[128] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave." in *Proceedings of the International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2018, pp. 451–470.

[129] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[130] "Intel Analysis of Speculative Execution Side Channels," https://kib.kiev.ua/x86docs/Intel/WhitePapers/336983-004.pdf, 2018.

[131] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," *Proc. ACM Program. Lang.*, vol. 5, jan 2021.

[132] X. Cheng, F. Tong, H. Wang, Z. Zhou, F. Jiang, and Y. Mao, "SpecLFB: Eliminating cache side channels in speculative executions," in *Proceedings of the 33st USENIX Security Symposium*, 2024.

[133] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SPECCFI: Mitigating Spectre attacks using CFI informed speculation," in *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P)*, 2020, pp. 39–53.

[134] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, "Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2023, pp. 1220–1237.

[135] J. Cabrera-Arteaga, N. Fitzgerald, M. Monperrus, and B. Baudry, "Wasm-mutate: Fast and effective binary diversification for WebAssembly," *Computers & Security*, vol. 139, 2024.

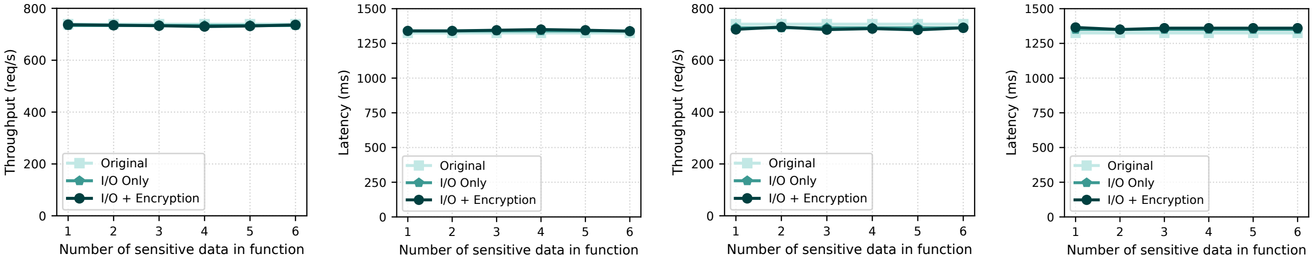
[136] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, and C. F. et al., "Going beyond the limits of SFI: Flexible and secure hardware-assisted in-process isolation with HFI," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2023.

Appendix

A. Stress Testing Experiments

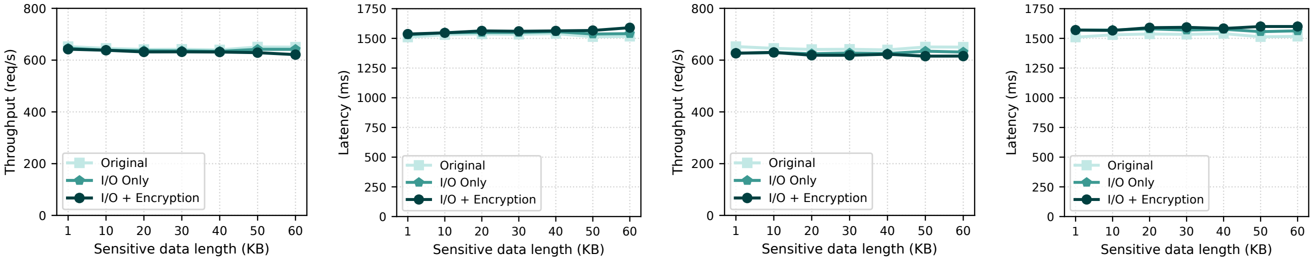
We performed some additional stress testing experiments to explore whether increasing the number of sensitive data objects and their size affects the performance overhead. We modified the "Authenticating Users at the Edge" application (Table III) to evaluate these two scenarios. Additionally, we removed the feature of storing data in the key–value store from this application to avoid the overhead associated with I/O-centric tasks.

1) Increasing the Number of Sensitive Data Objects: For this experiment, we modified the application and increased the number of sensitive environment variables to six, each 60 bytes in length. These variables' values are compared with those received from incoming requests, which have a length of 1KB. We chose to include six variables because, as shown in Figure 2, most serverless applications contain fewer than six



(a) Throughput (remote). (b) Latency (remote). (c) Throughput (local). (d) Latency (local).

Fig. 3: Throughput reduction and latency increase for an increasing number of sensitive data objects, when the I/O module runs on a different host (left) and on the same host with Spin (right), with a constant request size of 1KB.



(a) Throughput (remote). (b) Latency (remote). (c) Throughput (local). (d) Latency (local).

Fig. 4: Throughput reduction and latency increase for an increasing size of sensitive data, when the I/O module runs on a different host (left) and on the same host with Spin (right), with a constant request size of 60KB.

secrets. Initially, we measured the baseline overhead of the Spin framework and the I/O module when it operates on a different host (Remote) and on the same host as Spin (Local), specifying that none of the six variables are secrets. Subsequently, we increased the number of sensitive data items from one to six to measure any additional overhead.

As shown in Figure 3, increasing the number of sensitive data items does not incur any observable increase in overhead in either the local or the remote scenarios. This result is expected, as from a computation perspective, the only additional source of overhead comes from the additional cryptographic operations and data parsing for marshaling/unmarshaling operations, which is negligible.

2) Increasing the Size of Sensitive Data Objects: In this experiment, we evaluated the impact of significantly increasing the size of encrypted data on the latency and throughput overhead, compared to the small sizes of sensitive data objects typically found in real-world applications. We modified the application to include two variables: one labeled as sensitive and the other used for padding. The size of the protected data was increased from 1KB to 60KB, while adjusting the second variable to keep the total request size constant at 60KB.

Figure 4 shows how the increasing size of encrypted data impacts throughput and latency. Generally, we observe that encryption, whether applied in remote or local settings, does not significantly affect throughput or latency for secret data sizes up to 40KB. However, as expected, handling larger secret data resulted in higher overheads. Specifically, encrypting a 60KB secret value yielded the highest observed overhead, with a 4.4% reduction in throughput and a 4.8% increase in latency.

3) Increasing the Number of Concurrent Functions: In our performance evaluation, presented in Table III, we assessed the I/O module's overhead under stress-testing conditions where all functions handled sensitive data, corresponding to a worst-case scenario. This configuration allowed us to measure the maximum potential overhead. In practice, real-world environments comprise a mix of protected and unprotected functions, likely resulting in lower overhead. To validate this, we conducted a separate experiment in which the functions listed in Table III that handle sensitive data run alongside eight additional serverless functions that do not handle sensitive data. This configuration was informed by our compatibility analysis, which indicated that approximately 42% of all studied applications manage sensitive data.

We measured the overhead associated with full LeakLess protection ("I/O + Encryption") in both local and remote scenarios, comparing it against the baseline performance of the original Spin framework. As anticipated, we observed a reduction in overhead for both settings. Specifically, in the local scenario, the maximum latency increase was 1.8% and the maximum throughput decrease was 1.9%. Notably, in the remote scenario, no significant overhead was observed.