Efficiency without Tears: Securing Multilingual Programs with TRINITY

Hao Zhu   Baojian Hua*   Xinrong Lin   Yufei Wu
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
{hhxk, lxr1210, wuyf21}@mail.ustc.edu.cn   bjhua@ustc.edu.cn*

***Abstract*—Despite the fact that most real-world programs are developed in multiple languages in the era of data science, existing security techniques are still limited to single-language programs. Worse yet, languages designed for high-performance computing often ignore the necessary security checking in foreign function interfaces (FFI) to pursue supreme execution efficiency. In consequence, security flaws and vulnerabilities in these systems might cause security issues, defeating their efficiency benefits.**

In this paper, we present TRINITY, the first holistic infrastructure designed to effectively and efficiently secure the FFIs of multilingual programs in high-performance computing scenarios. TRINITY consists of two key components: 1) a privilege separation by memory isolation to protect memory; and 2) a pointer sanitizer to sanitize memory accesses by unsafe code. The privilege separation is based on the latest Intel MPK hardware primitives, and the pointer sanitization is based on an indirection table data structure managed in host language memory, storing important meta information about host data. We have designed and implemented a prototype of TRINITY for Julia-C multilingual programs and carried out extensive experiments to evaluate its effectiveness, performance, and usefulness on microbenchmarks and real-world applications from diverse yet representative domains, including heterogeneous computing, web servers, databases, and machine learning. The experimental results demonstrate that TRINITY effectively safeguards memory access from unsafe guest code with low overheads: 4.7% for OpenCL, 6.4% for JuliaDB, and 2.17% for Knet, respectively.

***Index Terms*—High Performance Computing, Memory Protection, Intel MPK, Privilege Separation**

I. INTRODUCTION

Multilingual programs are increasingly important and popular in the era of data science, largely due to their capabilities to offer both programming flexibilities and execution efficiencies simultaneously by leveraging the strengths of each language [1] [2]. For example, PyTorch [3], a mainstream machine learning framework, makes use of Python to implement the user-level APIs, and C/C++ for its kernel [4] [5]. Due to its technical advantages, recent studies have demonstrated that 82% systems are developed using a multilingual paradigm [6]. Given their increasing popularities and important roles in modern softwares, there is an urgent need to secure multilingual programs.

Despite this security need, it is, however, intrinsically difficult to secure multilingual programs. The key difficulty lies in the fact that, in multilingual programs, vulnerabilities often arise at the *boundaries* between the multiple languages, instead of within a single language. Worse yet, foreign function interfaces (FFIs) serving as the boundary often ignore necessary security enforcements, either to simplify the interfaces or to achieve high performance. For example, Python [41] provides Python-C FFIs to call C native code but without any security guarantees [9] [13]. As another example, Julia [7], a promising language designed for high-performance computing, provides a `ccall` [23] API, which has no security checking mechanism. Therefore, it is crucial to investigate a holistic security mechanism to secure multilingual programs.

To address the security issues in multilingual applications, a significant amount of studies have been conducted, on diverse languages combinations (*e.g.*, Python-C [2] [9] [10], Rust-C [11] [12] [42], Go-C [14], and Java-C [72] [73] [31]), by leveraging different protection mechanisms (*e.g.*, sandboxing [29] [32] [34], privilege separation [35] [36] [37], and hardware primitives [26] [56] [57] [77]). For example, POLYCRUISE [2] analyzes Python-C programs via dynamic program analysis, but lacks protection capability. As another example, SCONE [31], a secure container for Docker, uses the Intel SGX [77] to coarsely isolate the external functions, but bring considerable penalties.

Challenges. Unfortunately, while prior studies have made considerable progress towards securing multilingual programs, investigating a holistic protection technique for high-performance computing scenarios still face two technical challenges: 1) protection penalties; and 2) protection granularities. First, in high-performance computing scenarios, it is challenging to achieve high efficiency on large volume of data involved, due to the potential penalties a protection might bring. For example, Vx32 [32], a sandboxing-based protection, brought more than 30% overhead for data-intensive workloads. In addition, although existing studies have demonstrated that the overhead of the hardware-based protection is low on small volume of data [24] [42], it is still unknown whether this technology can be applied to date-intensive scenarios, because with the volume of data increases, the overhead caused by a protection might be accumulated significantly [17] [18].

Second, it is challenging to design a fine-grained protection technique at data structure granularity. The memory protection techniques proposed in prior studies are coarse-grained in nature to protect either the full memory or a group of memory pages. For example, the minimum granularity libmpk [24] can protect is one physical page. Other protections such as FFI security [21] [42] or control flow integrity (CFI) [28] [39] [40] are coarse-grained and specific to pre-defined data structures lacking flexibilities. Therefore, a fine-grained and data structure-aware memory protection technique is essential to address this challenge.

Our work. In this paper, to fill the gap, we present the *first* holistic infrastructure to enhance the FFI security in the high-performance computing scenarios effectively and efficiently. To achieve this goal, we propose a framework dubbed TRINITY, which consists of two key components: 1) a *privilege separation* by memory protection, to protect host code memory from unmanageable accesses from unsafe guest code; and 2) an *indirection table*, to deal with the unchecked memory accesses to host data structures from unsafe guest code. The privilege separation is based on the latest Intel Memory Protection Keys (MPK) [26], a hardware protection technology which outperforms over prior methods as it operates at user-space without entering the kernels. And the indirection table is a data structure we designed to store important meta information about host data structures, in separate memory page groups protected by MPK. Hence all accesses to these host data structures are sanity checked against the indirection table to guarantee that only legal ones are allowed.

Following these designs, we have taken Julia-C combination as a showcase to implement a software prototype. We have selected Julia for several important reasons: first, Julia is an emerging language designed for high-performance computing scenarios such as numerical computing and machine learning. Hence, its popularity and impact make our study more significant. Second, Julia introduced a novel FFI `ccall` to invoke native code. While `ccall` is simple to use and efficient, it does not provide any security protections. Hence, TRINITY can be used to close this security gap and demonstrate its security enhancement capabilities. Although we have showcased our approach with a prototype for Julia, our approach is general and suitable for other multilingual programs as well (as discussed in § VII).

With this software prototype, we have conducted extensive experiments to evaluate its effectiveness, performance, and usefulness. First, to evaluate the effectiveness of TRINITY, we applied TRINITY to microbenchmarks, and experiments results demonstrated that TRINITY is effective in protecting Julia memory from malicious native code such as illegal memory reading or writing. Second, to evaluate the performance of TRINITY, we applied TRINITY to macrobenchmarks with four large, real-world, widely-used Julia applications from diverse domains. And the performance overhead TRINITY introduced is less than 4.7% for OpenCL (a heterogeneous computing framework), 6.4% for JuliaDB (a high-performance database), and 2.17% for Knet (a machine learning library), respectively. For HTTP, the response delay is less than 10 nanoseconds for each Web requests. Finally, experimental results demonstrate TRINITY is easy to use, as TRINITY translates the target Julia applications automatically, thus no developer intervention or manual code rewriting is required.

Contributions. To the best of our knowledge, this work represents the first step towards understanding the FFI security issues in multilingual programs and proposing a systematic solution to secure them without sacrificing efficiency. To summarize, our work makes the following contributions:

- We conducted a systematic study of multilingual programs FFI security issues in the field of high-performance computing scenarios.
- We presented an infrastructure dubbed TRINITY and its prototype implementation, to secure multilingual programs FFI effectively and efficiently.
- We conducted extensive experiments to evaluate the effectiveness, performance, and usefulness of TRINITY.

Outline. The rest of this paper is organized as follows. Section II presents the background for this work. Section III presents the overall design of TRINITY and the threat model. Section IV and V presents the design and a prototype implementation, respectively. Section VI presents the evaluation we conducted, and Section VII discusses limitations as well as directions for future work. Section VIII discusses the related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, in this section, we present the background knowledge for this work: the Julia programming language (§ II-A), the Julia Foreign Function Interface (FFI) (§ II-B), and Intel Memory Protection Keys (MPK) (§ II-C).

A. Julia

Julia is an emerging programming language designed for high performance and data intensive computation. Since its first public release in 2012, Julia has been used successfully in a large spectrum of domains such as data visualization, parallel computing, data science, machine learning, and high-performance computing [27].

Julia is designed and evolved with three important goals: 1) flexibility to support dynamic programming; 2) data-intensive computing-oriented; and 3) efficiency. Guided by these goals, Julia builds upon the lineage of mathematical programming languages, but also borrows features from dynamic languages (*e.g.*, Lisp [20], Perl [38], Python [41], Lua [44], and Ruby [58]). To achieve high performance, Julia uses static type inference to eliminate potential runtime penalties.

Julia is gaining more popularity and becoming an increasingly important language in recent years. More than 1,500 colleges and universities are using and teaching Julia [15] [16]. In the meanwhile, more than 10,000 companies (*e.g.*, Google, Intel, Microsoft, and NASA [48]) around the world are using Julia to develop high-performance systems.

B. Julia FFI

Julia introduced a novel foreign function interface `ccall` [23] to invoke C/C++ libraries. Julia FFI design takes a no boilerplate philosophy: native functions can be called directly from Julia without any glue code, code generation, or compilation. Specifically, Julia FFI mechanism supports two-way interactions between Julia and native code: 1) the `ccall`, allowing Julia code to invoke native functions; and 2) the Julia C APIs [30], allowing native code to invoke Julia functions.

Julia invokes native code. Julia code can invoke functions in native code via `ccall` without additional encapsulation. For example, to invoke a native function `foo` in a shared library `lib`, Julia code can make the following `ccall`:

```
ccall(("foo", "libPath"), retT, (argTs, ), args)
```

where `libPath` stands for the absolute path containing the library `lib`, and `retT`, `argTs`, and `args` are function `foo`'s return type, argument types, and arguments, respectively.

Julia's JIT compiler generates the same binary code for `ccall` as it does for a native call, thus calling a native function does not incur any overhead [33]. Furthermore, by passing pointers to native code, Julia allows native code to access Julia memory directly. This allows data to be manipulated in-place, which is very efficient in scenarios such as machine learning in which large matrix calculations are indispensable.

Native code calls Julia. To enable native code to call Julia functions, Julia provides a set of so-called *Julia-C APIs* (or Julia APIs for short). For example, the following C code snippet presents a minimal function to execute a piece of Julia code via a specific Julia API `jl_eval_string()`.

```
jl_init(); /* setup Julia context */
jl_eval_string("println(sqrt(2.0))");/*Julia code*/
jl_atexit_hook(0); /* notify the termination */
```

Julia APIs, supporting diverse functionalities such as data conversion, memory management, and exception handling, are essential to integrate Julia code into a C/C++ project.

C. Intel MPK

To quick switch the access permission for memory pages, Intel introduced a hardware security feature called Memory Protection Keys (MPK) in 2015, which appears in the newest lines of CPUs such as Skylake. With MPK, users can modify the 32 bits per-thread `pkru` register by two user space non-privileged instructions `rdpkru` and `wrpkru`. Compared with existing memory page protection technologies, the key advantage of MPK is that it can directly manipulate the page table and translation lookaside buffer (TLB) in user mode without switching to kernel mode, resulting in higher efficiency.

Many software abstractions (*e.g.*, ERIM [22], and libmpk [24]) have been proposed to make incorporation of MPK hardware technology easier. For example, the relatively new libmpk abstraction provides a group of APIs supporting page manipulations such as permission switch, memory pages allocation, initialization, and free. To utilize libmpk, function `mpk_init` shall be used first to obtain all the hardware protection keys from the kernel and initialize their metadata. Then `mpk_mmap` allocates a page group for a virtual protection key. The function `mpk_munmap` destructs a page group by freeing a virtual key for the page group and unmaps all the pages in it. On top of these primitive operations, libmpk also provides useful heap management APIs such as `mpk_malloc` and `mpk_free`, so that a developer can create a customized heap management subsystems with page groups, to protect sensitive data in memory. Furthermore, such abstractions have direct support for multithreading by leveraging hardware virtual threads [24].

TABLE I: Protection technologies for different interactions.

Control Transfer	**Description**	**Protection**
Host → Host	Host code accessing host-allocated memory	NA
Host → Guest	Host code accessing guest-allocated memory	Privilege separation
Guest → Host	Guest code accessing host-allocated memory	Indirection table
Guest → Guest	Guest code accessing guest-allocated memory	NA

Intel MPK, along with these software abstractions, shows significant performance advantages. For example, libmpk has runtime overhead of 1% for a high frequency of switching permission, while provides over an X8 performance improvement over the traditional `mprotect` system call for process-level permission switch [24].

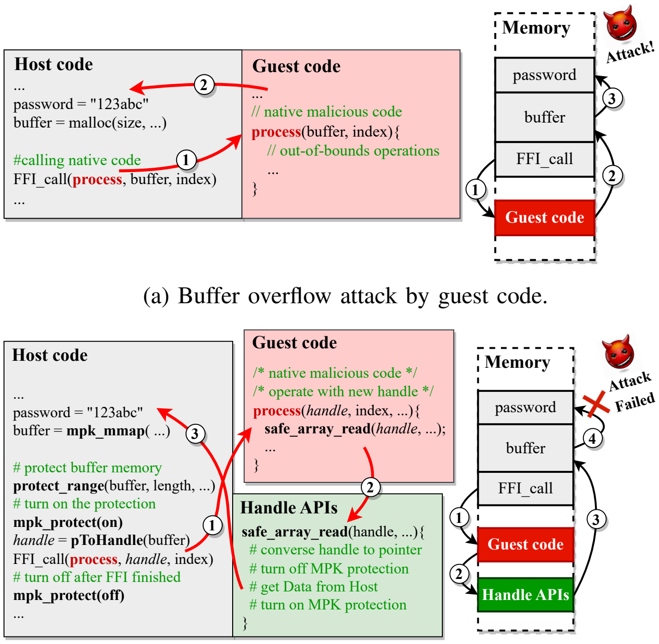
III. TRINITY OVERVIEW AND THREAT MODEL

In this section, we present an overview of how TRINITY works (§ III-A), then discuss the threat model (§ III-B).

A. Overall Design

Interactions. Based on the interactions and semantics of FFIs between host and guest code, we have identified, as TABLE I presented, four control transfers in a multilingual program: 1) control transfers between host code; as the threat model of this work (see Section III-B) specifies that host code is trusted, so no special protection technologies are required for this scenario; 2) control transfers from host code to guest code through FFI; as the guest code are not trusted, TRINITY utilizes privilege isolation to isolate the host memory to guarantee that guest code cannot perform unmanageable accesses; 3) control transfers from guest code to host; TRINITY utilizes the indirection table to guarantee that guest code can only access host memory in a fine-grained, controlled, and secure manner; and 4) control transfers between guest code; such transfers are out of the scope of this work and standard protection such as stack canary [84] or CFI [28] can be enforced. It is important to note that, although we have discussed these protection techniques separately for ease-of-presentation and understanding, they are really a unified approach in guaranteeing memory safety.

Design principles. After identifying all potential interactions, we present, in Fig. 1, the overall design for TRINITY. We have two important principles guiding the design of TRINITY: flexibility; and 2) automation. First, TRINITY should be flexible



Host code

```
...
password = "123abc"
buffer = malloc(size, ...)

#calling native code
FFI_call(process, buffer, index)
...
```

Guest code

```
...
// native malicious code
process(buffer, index){
  // out-of-bounds operations
   ...
}
```

(a) Buffer overflow attack by guest code.

Host code

```
...
password = "123abc"
buffer = mpk_mmap( ...)

# protect buffer memory
protect_range(buffer, length, ...)
# turn on the protection
mpk_protect(on)
handle = pToHandle(buffer)
FFI_call(process, handle, index)
# turn off after FFI finished
mpk_protect(off)
...
```

Guest code

```
/* native malicious code */
/* operate with new handle */
process(handle, index, ...){
  safe_array_read(handle, ...);
  ...
}
```

Handle APIs

```
safe_array_read(handle, ...){
  # converse handle to pointer
  # turn off MPK protection
  # get Data from Host
  # turn on MPK protection
}
```

(b) Defend against attack using TRINITY.

Fig. 1: The overall design of TRINITY: (a) depicts buffer overflow attacks by guest code using explicitly passed pointer parameters. (b) depicts the protection mechanisms of TRINITY, which include: 1) privilege separation: restricting access rights to protected memory pages before guest code execution; 2) pointer sanitizing: allowing guest code to access host memory only through trusted handle APIs.

enough to protect different interactions between host and guest code as control transfers.

Second, TRINITY should be fully automated thus minimal interventions are required from developers. Specifically, TRINITY is proposed to be used in two scenarios: 1) new projects development; or 2) legacy projects migration. In the first scenario, developers can integrate TRINITY into their projects without difficulty, as TRINITY provides an effective programming model and a group of easy-to-use APIs. In the second scenario, TRINITY provides automated tools to help developers migrate their legacy code (discussed in § V).

B. Threat Model

The focus of this work is on proposing a holistic protection for multilingual applications in high-performance computing. Therefore, we make the following assumptions in the threat model for this work.

We assume that the host environment, running the multilingual applications, has standard protections. For example, the underlying hardware or operating systems provide standard protections such as Data Execution Prevention (DEP) [83], Stack Canaries [84], and Address Space Layout Randomization (ASLR) [85]. Furthermore, the compiler has not been compromised by malicious adversaries so that the binaries generated from the compiler are trustworthy. Although operating systems and compilers security studies are very important, they are independent of and thus orthogonal to the study in this work. Furthermore, these research fields can also benefit from the research progress in this work.

We assume that host code is safe and will not pose a security threat to the application being investigated. For example, the host code does not trigger out-of-bounds buffers access, as every buffer access is checked against the buffer length. Thus, such an assumption is reasonable in reality.

We assume that the guest code is untrusted and unreliable. For example, if the native code function being called through FFI is vulnerable, adversaries can control the guest code to perform arbitrary attacks such as illegal memory reading or writing, or triggering buffer overflows. As our focus in this work is to study the FFI security, results in this work supplement classical guest code protections such as control-flow integrity (CFI) [28] [39] [40].

As the Intel MPK is a relatively new technology, thus we assume a latest line of Intel server-class CPU is available (Intel Skylake or newer). Furthermore, we assume that the software abstractions of library implementations of MPK (*e.g.*, ERIM, or libmpk), is secure. Although testing these abstractions or implementations are important research topic, it is irrelevant to and out of the scope of this work.

IV. TRINITY DESIGN

In this section, we present the design of TRINITY, by introducing the privilege separation by memory isolation (§ IV-A), and pointer sanitizing via the indirection table (§ IV-B).

A. Privilege Separation by Memory Isolation

We first present how TRINITY enforces privilege separation via memory isolation.

Page Groups. Intel MPK enforces memory protection based on *page groups*, all pages in one page group sharing the same protection key thus have the same privilege permissions. According to the Intel manual [25], newest line of Intel CPUs support up to 16 different page groups physically. The software abstractions of Intel MPK even support more virtual page groups, by protection key reusing in operating system kernels. For example, libmpk supports an unlimited number of page groups [24].

As TRINITY is designed to isolate memories of host and guest code, it splits the memory into two disjoint page groups: the host memory and the guest memory, as described by Fig. 1 (the gray and red region, respectively). TRINITY makes use of two distinct protection keys to protect the two page groups, respectively. Two important facts about this design choice deserve further explanation: 1) although it is possible to make use of more protection keys (thus more page groups), two is enough in this work for TRINITY to secure host FFIs; and 2) pages in one page group need not to be adjacent but may interleave. For example, a host memory page sits between two guest pages, which makes this protection strategy more flexible and convenient.

Privilege separation. Intel MPK enables setting up privilege permissions on pages in a page group simultaneously, and switching permissions quickly from user space without entering the kernel. TRINITY is designed to make use of two distinct groups of permissions to protect the two memory regions (the host memory and the guest memory), respectively. Host code has full access permissions to both memory regions, whereas the guest code only has permissions to access the guest memory.

The permissions change as following: 1) when the application starts executing, the host code sets up `"rw"` permissions for the two page groups, thus host code has full read/write access to the two memory regions; 2) when host transfers control to the guest code by calling any function in guest code, host first disables access to the host memory by clearing the `"rw"` permissions in the corresponding page group; 3) when guest code executes, it has full access to the guest memory but not the host memory; thus guest code has no way to disrupt host memory; and 4) when the control transfers back from guest code to host, host code enables the `"rw"` permissions again for the host memory.

It is important to note that since the MPK APIs is a user-level protection mechanism, that is, the `rdpkru` and `wrpkru` MPK instructions are non-privileged, so in theory, the guest code can also makes use of these instructions to switch host memory permissions, which breaks the security guarantees Intel MPK enforced. Thus, one must ensure that guest code does not switch the permissions of the host memory. To achieve this goal, one can leverage any standard binary scanning techniques [43] [45], to detect any MPK-related instructions in the target binary. The user is notified, if any such instructions exist. We will discuss further subtleties for this in § VII.

B. Pointer Sanitizing by Indirection Table

While the privilege separation mechanism discussed in the above section is effective in isolating host memory from guest memory, it is often overly restrictive in that it prohibits any legal access from the guest code, which is inconvenient in certain scenarios. For example, in a decompression application, host code may invoke a guest decompressing function `decomp()` through host FFI to achieve maximum efficiency. The guest function `decomp()` may need to access the original compressed data located in host memory. To make such interactions feasible, TRINITY introduced an indirection table to sanitize pointers in a fine-grained manner.

Specifically, this technology consists of four components: 1) the handle; 2) the indirection table; 3) the handle APIs; and 4) external function conversions; which are discussed next, respectively.

Handle. A handle is an abstract representation of concrete memory address, which is generated and managed by host code, and passed to guest code. The basic workflow a handle gets used is as follows: 1) for each memory address to be passed from host code to guest code, host code generates a fresh handle representing that address; 2) host code passes the generated handle, instead of raw address, to external guest code; 3) whenever guest code needs to access host memory, it invokes some host exposed *handle APIs*, passing the handle as arguments; the host code verifies the handle before accessing the memory the handle representing; and 4) when control transfers back from guest code to host code, host expires the corresponding handle passed, so that no other guest functions can use this handle any more.

Indirection table. TRINITY introduces an indirection table to record the mapping from a handle to the concrete memory address it represents. TRINITY makes use of the indirection table in the following manner: 1) host code creates an initially empty table t when the application starts; 2) when host code generates a handle h for a specific memory address a, TRINITY inserts the mapping $h \mapsto a$ into the table t; 3) when guest code access host memory by passing a handle h, host code looks up the indirection table t, for the address a the handle h corresponds to; and 4) after a guest function returns, the handle h that function uses is expired by removing h from the table t.

To guarantee memory safety by protecting the host memory effectively, the indirection table must satisfy four important requirements: first, the indirection table must be stored in host page groups thus is only accessible by host code; otherwise, suppose that the indirection table is stored in guest page groups, the guest code can access all memory stored in the table just by enumerating entries in the table; worse yet, guest code can insert fake addresses into the table to trigger subsequent arbitrary address read/write. Fortunately, by storing the indirection table in the host page groups protected by MPK, the guest code has no access to it.

Second, handles must be random enough thus be difficult to guess or forgery. Otherwise, suppose that an adversary can guess the value of a handle, then by looking up the indirection table with that guessed handle, the adversary is able to access a host memory address she had no permissions.

Third, to guarantee high efficiency, the indirection table should allow multithreaded concurrent accesses via reasonable protections such as a mutex.

Finally, as handles are extensively used by host FFIs, it should be fast to generate them and compare for equality; otherwise, they will incur considerable performance penalties.

Handle APIs. To allow guest code functions to use handles easily, TRINITY designed a group of *handle APIs*, which implemented in host and exposed to guest code. The handle APIs should include common operations on host, such as data structures manipulation, memory management, and exception handling.

Handle APIs supplement and enhance the standard C APIs. Specifically, the standard C APIs can be classified into two categories: 1) APIs that do not access host memory, these APIs can be used by guest code without any change, as they are memory safe; and 2) APIs accessing host memory, these should be replaced by a corresponding secure handle API.

External function conversion. host passes handles instead of raw memory addresses to guest code, which can be used to access host memory. To this end, guest code must be converted to reflect such changes: 1) each guest function accepting a raw address is converted to accept a handle representing that address; and 2) host APIs accessing host memory are converted to corresponding handle APIs.

While manual guest code conversion is possible, doing so is laborious and error-prone, especially for large multilingual projects with considerable guest code bases. To this end, an automatic technology is desired to perform such conversions in practice, which we will discuss in the next section.

V. Trinity Implementation

In this section, we present a prototype implementation of Trinity for Julia-C multilingual programs, by first introducing the implementation of privilege separation (§ V-A), and indirection table (§ V-B).

A. Implementation of Privilege Separation

Trinity leverages the libmpk library [24] to implement privilege separation. libmpk is a relatively new software abstraction for the Intel MPK technology, whose usefulness has been demonstrated by protecting real-world applications such as OpenSSL, JavaScript JIT compilers, and Memcached.

Implementation of page groups. Trinity implements two page groups: `GROUP_JULIA` and `GROUP_GUEST`, for the Julia memory and guest code memory, respectively. These two page groups are distinct integers to be used by libmpk functions. It is interesting to note that although the Intel CPU reserves 4 bits to represent the page groups (*i.e.*, the 32nd to 35th bits in the page table entry), indicating the valid page groups are in the range $(0, 16)$, libmpk supplies an infinite number of page groups by virtualizing the physical ones, simplifying the implementation.

Trinity utilizes the `mpk_mmap()` function, to allocate memory in corresponding page groups. For example, Trinity executes `addr = mpk_mmap(GROUP_JULIA, ..., perm, ...)` to allocates a chunk of memory in the `GROUP_JULIA` page groups, and assign the returning address to the variable `addr`. The `perm` are normally initialized to `PROT_READ|RROT_WRITE`, for read/write permissions.

Implementation of privilege separation. Trinity initialized read/write permissions for both the Julia memory and guest memory, so Julia code has full access to both memories initially. To enforce privilege separation, as Fig. 1 shows, Trinity disables the permissions of the Julia memory before each `ccall`, by calling `mpk_protect(GROUP_JULIA, PROT_NONE)`. Thus, when control transfers to guest code, guest code has no access to the Julia memory, which guarantees memory safety. When control transfers back from guest to Julia, Trinity restores permissions by calling `mpk_protect(GROUP_JULIA, PROT_READ|PROT_WRITE)`.

The libmpk library suggested a pattern to setup and clear permissions by leveraging `mpk_begin()` and `libmpk_end()` APIs. The key idea is, by wrapping the target code between these two functions, the target code can have the desired read/write permissions temporarily. However, we have observed, in implementing Trinity, that this pattern is infeasible, as the permissions are being disabled during the `ccall`, instead of being enabled.

TABLE II: The representative indirection table APIs.

API	Description
itable_new()	Create a new indirection table
itable_insert(itable, handle, addr)	Insert a mapping from handle to addr
itable_lookup(itable, handle)	Lookup a handle from the table
itable_remove(itable, handle)	Remove a handle from the table

B. Implementation of Indirection Table

Trinity makes use of the indirection table to sanitize pointers from guest code to Julia.

Implementation of handle. To achieve the design goals of handle randomness, fast generation, and equality comparison, Trinity's implementation makes use of 64 bit unsigned integers to represent handles. To guarantee randomness, Trinity makes use of the Random Module in Julia to generate strong pseudo random numbers. Comparing handles for equality is fast, as it is a primitive integer operation.

Although using more bits (*e.g.*, 128 bits), to represent a handle provides stronger randomness, but it makes passing handles to guest code difficult as most languages such as C/C++ do not support 128 bits primitive integers.

Implementation of indirection table. To support fast table retrieval, Trinity makes use of hash table data structures to implement a new datatype `itable` for the indirection table. Furthermore, Trinity supports these typical APIs as shown in TABLE II. These APIs have similar semantics to standard APIs in typical hash table libraries thus deserve no further explanations, for example, the `itable_remove()` function expires a valid handle, by removing the handle from the indirection table.

Trinity's current implementation will panic and exit for invalid handles, as such invalidity often indicates potential attacks. For example, if the `itable_lookup()` function fails to find the `handle` argument, it is possible that some adversary is trying to enumerate the indirection table with forged handles. However, Trinity also allows users to customize the implementation by supplying a user-defined error recovery routine. For example, a user-defined routine may generate a log, before skipping that operation.

Concurrency support. Table APIs are mutex-protected to allow safe concurrent accesses to the table by multiple threads. Specifically, to achieve maximum efficiency, Trinity supports a fine-grained concurrency, that is, each table entry is protected by a distinct mutex. Hence, multiple threads may execute different instances of the same table API with different handles (thus different mutex).

Implementation of Handle APIs. To allow external functions to access Julia memory securely through handles, Trinity implements a group of handle APIs, which provide read/write capabilities for Julia data structures. Trinity implements these APIs in Julia and exposes them to guest code. For example, the handle API handle_array_read in the following code snippet reads an element at index index from the array represented by the argument handle.

```
function handle_array_read(handle, index)
 addr = itable_lookup(itable, handle);
 assert(index>=0 && index<length(addr);
 return Int(addr[][index]);
end
```

Error checking and recovery code is omitted for simplicity. When guest code needs to access an array element, it transfers control to Julia by invoking the above function. The Julia code looks up the indirection table itable for the address addr of the array, before returning an array element at index.

External Function Conversion. Handles are used in two scenarios: developing new Julia projects; or migrating legacy ones. For the former, simply integrate the new handle APIs directly into native code development. The latter scenario is much more challenging, as legacy code must be converted to use the new handle APIs. We present, in Fig. 2, a simple yet illuminating sample. Two modifications are required to convert guest code: 1) the function argument is converted from a raw address array to an abstract handle (recall that TRINITY makes use of 64 bits integers to represent handles); and 2) the direct array access array[index] (line 3) is converted into invocations to the corresponding handle API handle_array_read, passing handle and index as arguments (line 7).

```
1  # invoke C function 'read_array'
2  ccall((:read_array, "./C_Lib.o"),Int32,(Ptr{
       UInt8},), array)
```

(a) The ccall function in Julia.

```
1  /* Guest function before conversion */
2  int read_array(int *array, int index)
3   { return array[index];}
4  /* Guest function after conversion */
5  int read_array(long long handle, int index){
6   jl_value_t *r = jl_call2(
7      handle_array_read, handle, index);
8   return jl_unbox_int32(r);}
```

(b) Before and after a guest function read_arrayis converted.

Fig. 2: Sample code illustrating how a guest function is converted.

To realize this process, we developed a prototype translator in TRINITY to convert the guest code automatically, by leveraging the CIL infrastructure [91]. Specifically, the conversion works in three major steps: 1) the guest code sources are parsed into abstract syntax trees (AST), a compiler internal data structure suitable for manipulation; 2) TRINITY automatically rewrites target functions, identified by function names in the Julia ccall API, to convert their arguments and bodies as discussed above. This rewriting process is essentially a one-pass syntax-directed AST conversion, which has been shown to be quite efficient in practice; and 3) TRINITY generates converted guest code by outputting the rewritten AST.

TABLE III: Macrobenchmarks of 4 real-world applications.

Application	Domain	LoC	Github Stars
OpenCL [46]	High-performance Computing	10,764	252
HTTP [47]	Web	15,549	592
JuliaDB [49]	Database	5,993	765
Knet [52]	Machine Learning	65,581	1,404

VI. EVALUATION

In this section, we present experiments to evaluate TRINITY.

A. Research Questions

By presenting the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. As TRINITY is proposed to secure multilingual programs, is it effective in protecting host programs against malicious guest code?

RQ2: Performance. As TRINITY makes use of latest Intel MPK and indirection table to enforce the protection, what is the performance of it?

RQ3: Usefulness. As TRINITY is proposed to secure practical programs, is it useful to large and real-world applications?

B. Experimental Setup

All experiments and measurements are performed on a server equipped with a 4-physical Intel i7 core (8 hyperthread) CPU and 16 GB of RAM running Ubuntu 20.04. The Julia compiler version is v1.8.0.

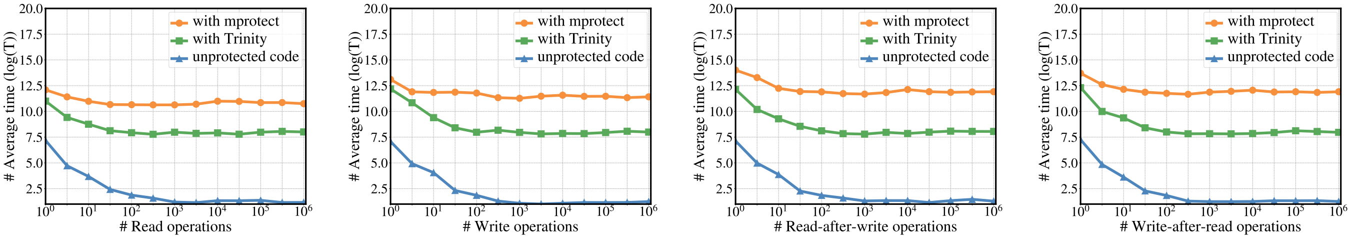
C. Datasets

To conduct the evaluation, we selected and created two datasets: microbenchmarks, and macrobenchmark of real-world applications.

Microbenchmarks. Evaluating the effectiveness of a protection technique (TRINITY in this work) requires a multilingual dataset that comes with ground truth. However, such a dataset, to the best of our knowledge, does not exist. Hence, we take the first step to manually create such a dataset dubbed **JBench**, for evaluating protection techniques, including but not limited to TRINITY. Currently, **JBench** consists of 10 programs, 4 of which are used to verify the effectiveness of privilege separation and indirection table, 6 of which to measure performance. We are continuing to maintain and augment this benchmark by including more programs.

Real-world applications. For better benchmark representation, we choose real-world applications that are: 1) high-performance computing-related; 2) multilingual (*i.e.*, with ccall); 3) frequently updated and maintained; and 4) popular (more than 200 Github stars).

According to these selection criteria, we present, in TABLE III, four real-world applications from diverse domains: 1) OpenCL [46], a popular package offering a complete solution



(a) Single Read

(b) Single Write

(c) Read-after-Write

(d) Write-after-Read

Fig. 3: Performance evaluation results of privilege separation for four scenarios: read, write, read-after-write, and write-after-write.

for heterogeneous computing in Julia. OpenCL will invoke, via `ccall`, specific guest functions to complete some core functionalities such as context construction, events triggering, and computation initiation. 2) HTTP [47], an open source and prevalent software package for developing client or server web applications in Julia. Julia HTTP makes extensive use of `ccall`, to invoke guest functions finishing critical tasks such as initializing sockets, manipulating configuration strings, or reusing server ports. 3) JuliaDB [49], a widely-used database in Julia [50]. JuliaDB invokes, through `ccall`, guest functions to perform key operations such as loading data, or reading parameters from the memory pool. 4) Knet [52], an important deep learning framework in in Julia, ranking second among all Julia AI packages [53]. In its model training phase, Knet invokes `ccall`, to call guest functions to finish tensor operations such as reduction, broadcast, deepcopy, and deepmap.

D. RQ1: Effectiveness

To answer **RQ1** by investigating the effectiveness of TRINITY, we conducted experiments to testify the two protection mechanisms (*i.e.*, privilege separation, and indirection table), by evaluating TRINITY against the microbenchmark **JBench**.

First, we construct a group of microbenchmarks consisting of multilingual Julia applications, and manually inject common kinds of vulnerabilities into the guest code of these applications. For example, we inject arbitrary memory accesses, that is, by casting an arbitrary integer into a pointer, the guest code can access any address in the Julia memory. We also inject buffer overflows, that is, by writing passed the end of an array in the Julia memory, the guest code can overwrite data stored in Julia memory.

After injecting these vulnerabilities, we first compiled and executed these benchmarks, and observed that all of them crashed by triggering memory segment faults. Then, we applied TRINITY to these benchmarks, compiled and executed them for a second time, we observed TRINITY successfully detected all of these attacks and reported informative information. For example, for arbitrary memory access, the Intel MPK reports the required permissions are missing; and for buffer overflow, the indirection table enforced array accesses are always in range.

Summary: The experimental results demonstrate that TRINITY is effective in protecting multilingual programs from common types of vulnerabilities in the guest code.

E. Performance

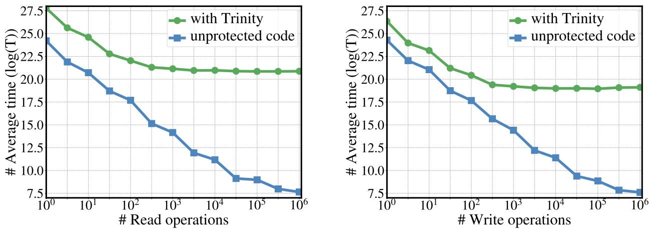
To answer **RQ2** by investigating the performance and overhead TRINITY introduced, we testified micro benchmarks for four memory access scenarios: 1) read-only; 2) write-only; 3) read-after-write; and 4) write-after-read. We evaluated privilege separation and the indirection table separately, to gain a thorough understanding of the performance penalty.

Privilege Separation Performance. As Fig. 3 shows, we executed three different versions for each benchmark: 1) the original one (blue); 2) the one with TRINITY (green); and 3) the one with `mprotect` (orange). In each sub-figure, the x-axis stands for the number of operations performed, from 1 to 10^6; and the y-axis gives the average running time for the corresponding operations, in nanoseconds. Furthermore, to make the difference between running time clearer, we have normalized the average running time by $y = \log(T)$,

We observed that, for the four scenarios, the average overhead TRINITY introduced was 600 nanoseconds (recall that 1 nanosecond = 10^{-9} second) to the running time on average, which is tiny and in par with prior studies on MPK. Compared with MPK, the `mprotect` protection is of much more significant overhead: for the read-only and write-only operations, the average overhead are 1,600 and 2,800 nanoseconds, respectively. And for read-after-write and write-after-read operations, the overhead are 4,600 and 4,100 nanoseconds, respectively.

Indirection Table Performance. To investigate the overhead introduced by the indirection table, we developed a multilingual Julia benchmark which access arrays located in Julia memory. With this baseline, we created a safe version of this benchmark by replacing all pointer parameters by handles, as well as replacing direct manipulation of pointers by indirect handle APIs invocations.

We then compiled and executed these two benchmarks. For each operation being evaluated, we executed 1 to 10^6 rounds to calculate the average running time. Fig. 4 shows the experiment results. In each figure, the x-axis presents the



(a) Single Read

(b) Single Write

Fig. 4: Performance evaluation results of indirection table for two scenarios: single-read and single-write.

numbers of operation, and y-axis presents the average running time (again in logarithmic time).

For read and write operations, the indirection table adds 7,100 and 23,000 nanoseconds on an average, respectively. This overhead is practical and acceptable for two reasons: 1) it is in par with prior studies on runtime sanitizers; and 2) this performance penalty only exists on those guest functions that invoke handle APIs, while others do not involve this "pay-as-you-go" penalty.

Summary: The experimental results demonstrate that TRINITY is efficient and has significantly lower performance overhead compared to the Linux system service `mprotect`.

F. Usefulness

To answer **RQ3** by demonstrating the usefulness of TRINITY, we applied TRINITY to four large and real-world Julia applications from 4 representative fields: heterogeneous computing, web servers, database, and machine learning. It should be noted that the effectiveness of TRINITY was verified on these four applications by injecting memory attacks intentionally. Thus, we focus on its performance in this section.

1) **Heterogeneous Computing:** We modified OpenCL by introducing TRINITY and recompiled it to obtain a safe version `OpenCL.safe` to compare it with the original version `OpenCL.raw`. Then, we developed two identical applications to accelerate tensor additions using GPUs. To investigate the effect of different tensor sizes, we increased tensor sizes from 4KB to 4×10^1KB, 4×10^2KB, 4×10^3KB, 4×10^4KB, and 4×10^5KB. As a result, the number of physical pages storing these tensors increased from 1 to 10^1, 10^2, 10^3, 10^4, and 10^5, respectively. We recorded the total execution time for different data size, and obtained the average time spent per physical page by dividing page numbers.

Fig. 5a presents the experimental results. With the page numbers increasing, the average time per page for both versions of OpenCL decrease. And the overhead TRINITY introduced eventually stabilizes at below 4.7%.

2) **Web Servers:** We compiled the HTTP with and without TRINITY to two binaries `Http.safe` and `Http.raw`, respectively. We then deployed two Web servers responding `GET` requests from clients. Next, we recorded the average responding time of the two Web servers for 10 rounds, each with 10,000 requests, respectively.

Fig. 5b presents the experimental results. The average response delay (that is, the overhead) for Web requests is less than 10 nanoseconds on average. It should be noted that when the business of the Web server becomes more complex, the proportion of the additional overhead will decrease further.

3) **Database:** We applied TRINITY to JuliaDB and compiled the source code to obtain two binaries, `JuliaDB.safe` and the original `JuliaDB.raw`. Then, we ran the two Database binaries on Hflights.csv [51] data source. In experiments, we performed 4 operations on the data: filter, reorder, reindex, and function applying. We recorded the average execution time for each operation.

Fig. 5c presents the experimental results. The performance overhead introduced by TRINITY for the four operations are 2.86%, 3.80%, 6.39%, and 2.73%, respectively.

4) **Machine Learning:** We revised and recompiled Knet to obtain two versions of the library: `Knet.safe` and `Knet.raw`. Then, we wrote two digit recognition applications with MNIST [54] based on LeNet [55], and used TRINITY to protect the training data in `Knet.safe`. In the experiment, we set the epoch of model training from 2 to 10, then record the training time of these two applications, respectively.

Fig. 5d presents the experimental results. The overhead introduced by TRINITY is 2.17% on average. Furthermore, TRINITY affects neither the accuracy of the model, nor the training data.

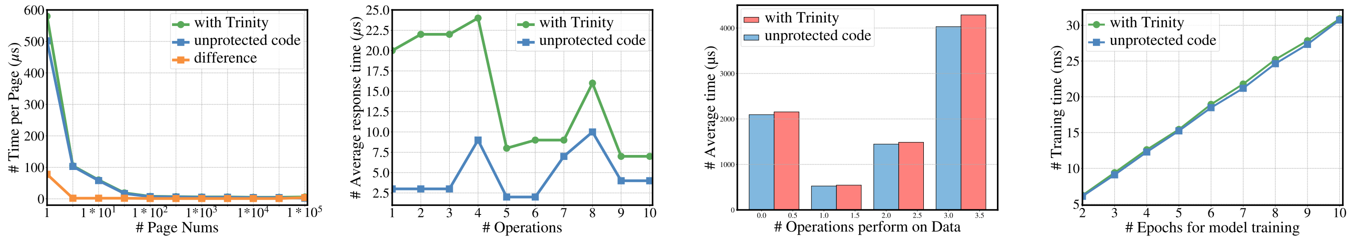
Summary: The experimental results of applying TRINITY to 4 real-world applications show that TRINITY is useful for securing large real-world multilingual programs, with insignificant runtime overhead.

VII. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work. It should be noted that this work represents the first step towards proposing an effective protection for securing multilingual programs.

Binary analysis. While Intel MPK is very efficient, by allowing processes permission switch at user space with two non-privileged MPK instructions `rdpkru` and `wrpkru`, it also allows a malicious attacker to abuse these instructions to switch permissions. In this work, we have leverage static binary analysis to scan the native code for these instructions, following existing studies. Although we do not encounter in our experiments, some systems (*e.g.*, Just-In-Time compilers [86] [87], or Self-Modifying-Code [88] [89] [90]) may generate code at runtime, posing challenges for detection. One promising approach to address this challenge is to perform dynamic code analysis [64] [65] [66] [67], and we leave this as our future work.

Error injection. To evaluate the effectiveness of TRINITY, we manually injected specific memory errors into the guest code of target multilingual Julia applications. The experimental results demonstrated that this approach is effective in performing



(a) OpenCL: Average execution time spent per memory page.The difference indicates the average overhead introduced.

(b) HTTP: Average response time of two Web servers for 10 rounds of testing.

(c) JuliaDB: Average time for four DB operations on two database containing same dataset.

(d) Knet: Average training time of two models at different epochs, with the same net and dataset.

Fig. 5: TRINITY performance evaluation results for OpenCL (Heterogeneous Computing), HTTP (Web Application), JuliaDB (DataBase), and Knet (Machine Learning).

the evaluations on microbenchmarks. To conduct experiments on Julia applications in the wild, an automatic method is desired. To this end, one promising approach is automatic error injection tools [68] [69], which may minimize the manual efforts required.

Supporting other languages. While we focus on the Julia-C as our showcase in this work due to the increasing popularity and importance of Julia, real-world multilingual system may contain other languages as well. Supporting other languages with TRINITY is straightforward, as the design (§ IV) is neutral to specific languages. In the near future, we plan to apply TRINITY to Python-C, another popular language in data science.

Hardware primitives. Intel MPK is a relatively new memory protection technology, which was used in this paper to protect FFIs. Similar to Intel MPK, other hardwares mechanisms (*e.g.*, IBM Storage Protection [56] or ARM Domains [57]), provide memory key protection as well. We believe the technique presented in this paper can also leverage these hardwares due to the similarity between hardware features, and we leave it a future work to extend TRINITY to other hardwares.

VIII. Related Work

Native code security. A lot of research efforts have been devoted to native code security. Necula et al. [59] presented CCured guaranteeing type safety for legacy C programs. Jim et al. [60] proposed Cyclone as a safe dialect of C. Wang et al. [61] proposed a polymorphic SSP (P-SSP) technology to re-randomize the canaries. Jang et al. [62] propose a technology of code replacement to prevent buffer overflows. Ren et al. [70] proposed neural network models to detect buffer overflow vulnerabilities. A key difference between these studies and the work in this paper, is that we focus on securing FFIs, hence, these studies are orthogonal to and thus supplement TRINITY.

Foreign function interface security. The FFI security has been extensively studied. Rivera et al. [42] proposed a framework Galeed to secure Rust FFIs. Terei et al. [80] [81] presented Safe Haskell to securely executes arbitrary unsafe code in Haskell. Furr et al. [71] proposed a multilingual type inference system to check OCaml FFIs, which was further extended to check Java Native Interface (JNI) programs [72]. Tan et al. [73] proposed the SafeJNI framework to guarantee type safety for JNI programs, which was further extended to native code in JDK [77]. Hu et al. [19] studied Python-C API security. A major limitation of existing studies is that, unlike our work, they have not systematically investigated a holistic infrastructure to secure multilingual programs in data intensive computing scenarios, which are increasingly important in the era of data science.

Hardware primitives for memory protection. Many hardware primitives have been proposed to protect memory. Intel proposed SGX [77] instruction set extension, which implements hardware-based enclave container to provide confidentiality and integrity protection for code and data. Intel MPK [26] added specific physical registers as well as special instructions to reduce the overhead of switching page access permissions. Similar to Intel MPK, IBM proposed the Storage Protection primitive [56]. The ARM platform provide several techniques, such as ARM Domains [57]. Shreds [78], ARMlock [79], and FlexDroid [82], to isolate insecure code or third-party libraries from sensitive data. However, the focus of this work is not to introduce new hardware protection primitives, but to enhance multilingual programs security by leveraging the latest hardware protection primitives.

IX. Conclusion

This paper presented TRINITY, the first holistic infrastructure to secure multilingual programs without sacrificing efficiency. We utilized a latest hardware protection primitive MPK to implement privilege separation, and a novel indirection table data structure to sanitize pointers from untrustworthy guest code. We have implemented a software prototype for TRINITY targeting the Julia-C which is becoming increasingly important. Experimental results demonstrated TRINITY is effective, efficient, and useful. Overall, the work in this paper represents a first step towards securing the multilingual FFIs, making languages designed for high-performance computing more secure without sacrificing efficiency.

REFERENCES

[1] Li W, Li L, Cai H. On the vulnerability proneness of multilingual code[C]//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022: 847-859.

[2] Li W, Ming J, Luo X, et al. PolyCruise: A Cross-Language Dynamic Information Flow Analysis[C]//31st USENIX Security Symposium (USENIX Security 22). 2022: 2513-2530.

[3] The PyTorch platform. Retrieved May 31,2023 from https://pytorch.org/.

[4] The PyTorch platform. Retrieved May 31,2023 from https://github.com/pytorch/pytorch.

[5] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library[J]. Advances in neural information processing systems, 2019, 32.

[6] Yang H, Li W, Cai H. Language-agnostic dynamic analysis of multilingual code: promises, pitfalls, and prospects[C]//Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2022: 1621-1626.

[7] The Julia language. Retrieved May 31,2023 from https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code/.

[8] GitBook: Lua sandbox library(1.4.0). Retrieved May 31,2023 from https://github.com/mozilla-services/lua_sandbox.

[9] Hu M, Zhang Y. The Python/C API: evolution, usage statistics, and bug patterns[C]//2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020: 532-536.

[10] Lin X, Hua B, Fan Q. On the Security of Python Virtual Machines: An Empirical Study[C]//2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2022: 223-234.

[11] Wang H, Wang P, Ding Y, et al. Towards memory safe enclave programming with rust-sgx[C]//Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. 2019: 2333-2350.

[12] Wan S, Sun M, Sun K, et al. RusTEE: developing memory-safe ARM TrustZone applications[C]//Annual Computer Security Applications Conference. 2020: 442-453.

[13] Li S, Tan G. Finding reference-counting errors in Python/C programs with affine analysis[C]//ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28. Springer Berlin Heidelberg, 2014: 80-104.

[14] Ding B, Zhang Y, Chen J, et al. CGORewritter: A better way to use C library in G[C]//2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2023: 688-692.

[15] Development of Julia Language. Retrieved May 31,2023 from https://juliacomputing.com/media/2022/02/julia-turns-ten-years-old/.

[16] Julia Growth Statistics. Retrieved May 31,2023 from https://juliacomputing.com/blog/2022/01/newsletter-january/.

[17] Coppolino L, D'Antonio S, Mazzeo G, et al. A comparative analysis of emerging approaches for securing Java software with Intel SGX[J]. Future Generation Computer Systems, 2019, 97: 620-633.

[18] Priebe C, Muthukumaran D, Lind J, et al. SGX-LKL: Securing the host OS interface for trusted execution[J]. arXiv preprint arXiv:1908.11143, 2019.

[19] M. Hu and Y. Zhang, "The Python/C API: Evolution, Usage Statistics, and Bug Patterns," 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 2020, pp. 532-536, doi: 10.1109/SANER48275.2020.9054835.

[20] The Lisp language. Retrieved May 31,2023 from https://lisp-lang.org/.

[21] Java Native Interface. Retrieved May 31,2023 from https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[22] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In 28th USENIX Security Symposium (USENIX Security 19).

[23] Julia FFI API. Retrieved May 31,2023 from https://docs.julialang.org/en/v1/base/c/#ccall

[24] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In 2019 USENIX Annual Technical Conference (USENIX ATC 19).

[25] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2018.

[26] Burow N, Zhang X, Payer M. SoK: Shining light on shadow stacks[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 985-999.

[27] The Julia language. Retrieved May 31,2023 from https://julialang.org/.

[28] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security (TISSEC), 2009, 13(1): 1-40.

[29] Prevelakis V, Spinellis D. Sandboxing Applications[C]//USENIX Annual Technical Conference, FREENIX Track. 2001: 119-126.

[30] Embedding Julia in C. Retrieved May 31,2023 from https://docs.julialang.org/en/v1/manual/embedding/.

[31] Arnautov S, Trach B, Gregor F, et al. Scone: Secure Linux containers with Intel SGX[C]//OSDI. 2016, 16: 689-703.

[32] Ford B, Cox R. Vx32: lightweight user-level sandboxing on the x86[C]//USENIX Annual Technical Conference. 2008: 293-306.

[33] Julia's JIT Compiler. Retrieved May 31,2023 from https://docs.julialang.org/en/v1/manual/embedding/.

[34] Dewald A, Holz T, Freiling F C. ADSandbox: Sandboxing JavaScript to fight malicious websites[C]//proceedings of the 2010 ACM Symposium on Applied Computing. 2010: 1859-1864.

[35] Seo J, Kim D, Cho D, et al. FLEXDROID: Enforcing In-App Privilege Separation in Android[C]//NDSS. 2016.

[36] Yu C, Li L X, Wang K, et al. Protecting the security and privacy of the virtual machine through privilege separation[C]//Applied Mechanics and Materials. Trans Tech Publications Ltd, 2013, 347: 2488-2494.

[37] Brumley D, Song D. Privtrans: Automatically partitioning programs for privilege separation[C]//USENIX Security Symposium. 2004, 57(72).

[38] The Perl language. Retrieved May 31,2023 from https://www.perl.org/.

[39] Göktas E, Athanasopoulos E, Bos H, et al. Out of control: Overcoming control-flow integrity[C]//2014 IEEE Symposium on Security and Privacy. IEEE, 2014: 575-589.

[40] Carlini N, Barresi A, Payer M, et al. Control-flow bending: On the effectiveness of control-flow integrity[C]//24th USENIX Security Symposium (USENIX Security 15). 2015: 161-176.

[41] The Python language. Retrieved May 31,2023 from https://www.python.org/.

[42] Rivera E, Mergendahl S, Shrobe H, et al. Keeping safe rust safe with galeed[C]//Annual Computer Security Applications Conference. 2021: 824-836.

[43] Hovav Shacham et al. 2007. The Geometry of Innocent Flesh on the Bone: Returninto-libc without Function Calls (on the x86). In ACM conference on Computer and communications security (CCS).

[44] The Lua language. Retrieved May 31,2023 from https://www.lua.org/.

[45] Ford B, Cox R. Vx32: lightweight user-level sandboxing on the x86[C]//USENIX Annual Technical Conference. 2008: 293-306.

[46] OpenCL.jl. Retrieved May 31,2023 from https://github.com/JuliaGPU/OpenCL.jl.

[47] HTTP.jl. Retrieved May 31,2023 from https://juliahub.com/ui/Packages/HTTP/zXWya/0.9.17.

[48] Prevelant of Julia. Retrieved May 31,2023 from Retrieved May 31,2023 from https://www.infoq.cn/article/b0dpmasunf3lbb8y2svq.

[49] JuliaDB. Retrieved May 31,2023 from https://juliadb.org/.

[50] Rank of JuliaDB. Retrieved May 31,2023 from https://juliapackages.com/packages?search=database.

[51] hflights.csv. Retrieved May 31,2023 from https://github.com/selva86/datasets/blob/master/hflights.csv.

[52] Knet.jl. Retrieved May 31,2023 from https://github.com/denizyuret/Knet.jl.

[53] Rank of Knet. Retrieved May 31,2023 from https://juliapackages.com/c/ai?sort=stars.

[54] mnist. Retrieved May 31,2023 from http://yann.lecun.com/exdb/mnist/.

[55] LeCun Y, Boser B, Denker J S, et al. Backpropagation applied to handwritten zip code recognition[J]. Neural computation, 1989, 1(4): 541-551.

[56] IBM Storage Protection. Retrieved May 31,2023 from https://www.ibm.com/products/storage-protect.

[57] ARM. ARM® Architecture Reference Manual ARMv7- A and ARMv7-R edition, 2018.

[58] The Ruby language. Retrieved May 31,2023 from https://www.ruby-lang.org/en/.

[59] Necula G C, Condit J, Harren M, et al. CCured: Type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems (TOPLAS), 2005, 27(3): 477-526.

[60] Jim T, Morrisett J G, Grossman D, et al. Cyclone: a safe dialect of C. USENIX Annual Technical Conference, General Track. 2002: 275-288.

[61] Wang Z, Ding X, Pang C, et al. To detect stack buffer overflow with polymorphic canaries[C]//2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2018: 243-254.

[62] Young-Su Jang, Jin-Young Choi: Automatic Prevention of Buffer Overflow Vulnerability Using Candidate Code Generation. IEICE Trans. Inf. Syst. 101-D(12): 3005-3018 (2018).

[63] LLVM. Retrieved May 31,2023 from https://llvm.org/.

[64] Bayer U, Moser A, Kruegel C, et al. Dynamic analysis of malicious code[J]. Journal in Computer Virology, 2006, 2(1): 67-77.

[65] Ball T. The concept of dynamic analysis[C]//Software Engineering—ESEC/FSE'99. Springer, Berlin, Heidelberg, 1999: 216-234.

[66] Tzermias Z, Sykiotakis G, Polychronakis M, et al. Combining static and dynamic analysis for the detection of malicious documents[C]//Proceedings of the Fourth European Workshop on System Security. 2011: 1-6.

[67] Poeplau S, Fratantonio Y, Bianchi A, et al. Execute this! analyzing unsafe and malicious dynamic code loading in android applications[C]//NDSS. 2014, 14: 23-26.

[68] Kanawati G A, Kanawati N A, Abraham J A. FERRARI: A flexible software-based fault and error injection system[J]. IEEE Transactions on computers, 1995, 44(2): 248-260.

[69] Cho H, Mirkhani S, Cher C Y, et al. Quantitative evaluation of soft error injection techniques for robust system design[C]//Proceedings of the 50th Annual Design Automation Conference. 2013: 1-10.

[70] Ren J, Zheng Z, Liu Q, et al. A buffer overflow prediction approach based on software metrics and machine learning[J]. Security and Communication Networks, 2019, 2019.

[71] Furr M, Foster J S. Checking type safety of foreign function calls[J]. ACM SIGPLAN Notices, 2005, 40(6): 62-72.

[72] Michael Furr, Jeffrey S. Foster: Polymorphic Type Inference for the JNI. ESOP 2006: 309-324.

[73] Tan G, Appel A W, Chakradhar S, et al. Safe Java native interface[C]//Proceedings of IEEE International Symposium on Secure Software Engineering. 2006, 97: 106.

[74] Li S, Tan G. Finding bugs in exceptional situations of JNI programs[C]//Proceedings of the 16th ACM conference on Computer and communications security. 2009: 442-452.

[75] Li S, Tan G. JET: exception checking in the java native interface[J]. ACM SIGPLAN Notices, 2011, 46(10): 345-358.

[76] Li S, Tan G. Exception analysis in the java native interface[J]. Science of Computer Programming, 2014, 89: 273-297.

[77] Costan V, Devadas S. Intel sgx explained[J]. IACR Cryptol. ePrint Arch., 2016, 2016(86): 1-118.

[78] Chen Y, Reymondjohnson S, Sun Z, et al. Shreds: Fine-grained execution units with private memory[C]//2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016: 56-71.

[79] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS), Scottsdale, Arizona, November 2014.

[80] Terei D, Marlow S, Peyton Jones S, et al. Safe haskell[C]// Proceedings of the 2012 Haskell Symposium. 2012: 137-148.

[81] Gill A. Type-safe observable sharing in Haskell[C]//Proceedings of the 2nd ACM SIGPLAN symposium on Haskell. 2009: 117-128.

[82] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FlexDroid: Enforcing In-App Privilege Separation in Android. In Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, February 2016.

[83] Microsoft. Description of the Data Execution Prevention (DEP) feature. Retrieved May 31,2023 from https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[84] Cowan C, Beattie S, Day R F, et al. Protecting systems from stack smashing attacks with StackGuard[C]//Linux Expo. 1999.

[85] PaX Address Space Layout Randomization. Retrieved May 31,2023 from https://pax.grsecurity.net/docs/

[86] Frassetto T, Gens D, Liebchen C, et al. Jitguard: hardening just-in-time compilers with sgx[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2405-2419.

[87] Cramer T, Friedman R, Miller T, et al. Compiling Java just in time[J]. Ieee micro, 1997, 17(3): 36-43.

[88] Cai H, Shao Z, Vaynberg A. Certified self-modifying code[C]//Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2007: 66-77.

[89] Giffin J T, Christodorescu M, Kruger L. Strengthening software self-checksumming via self-modifying code[C]//21st Annual Computer Security Applications Conference (ACSAC'05). IEEE, 2005: 10 pp.-32.

[90] Anckaert B, Madou M, De Bosschere K. A model for self-modifying code[C]//International Workshop on Information Hiding. Springer, Berlin, Heidelberg, 2006: 232-248.

[91] Necula G C, McPeak S, Rahul S P, et al. CIL: Intermediate language and tools for analysis and transformation of C programs[C]//International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 2002: 213-228.