Efficiency without Tears: Keeping Julia Safe with SAFEJ

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

***Abstract*—Julia is an emerging programming language targeting numerical analysis and data intensive computation, and has been used successfully in many fields such as data visualization, parallel computing, machine learning, and high-performance computing. To improve performance and reuse legacy code, Julia introduced a foreign function interface (FFI) called `ccall`, to interact with external C/Fortran code. Unfortunately, `ccall` lacks necessary security mechanisms and protections, which makes multilingual Julia applications vulnerable and exploitable. This paper presents SAFEJ, an infrastructure for improving Julia FFI security. SAFEJ consists of two key components: 1) a privilege separation by memory isolation to protect Julia memory; and 2) an indirection table, to sanitize memory accesses by unsafe native code. The privilege separation is based on the latest Intel MPK hardware protection technology. The indirection table is a data structure hosted in Julia memory storing important meta information about Julia data structures. We designed and implemented a prototype for SAFEJ, and conducted extensive experiments to evaluate its effectiveness, performance, and usefulness on micro and macro-benchmarks. Experimental results demonstrated that SAFEJ is effective in protecting Julia memory from unsafe native code. The usefulness and efficiency of SAFEJ are demonstrated by applying it on real-world large Julia applications from 4 representative fields including OpenCL for heterogeneous computing, HTTP for Web servers, JuliaDB for Database, and Knet for machine learning. The average overhead SAFEJ introduced is 4.7% for OpenCL, 6.4% for JuliaDB, and 2.17% for Knet, respectively. The average extra response delay for HTTP is less than 10 nanoseconds.**

***Index Terms*—Julia FFI, Memory Protection, Intel MPK, Privilege Separation**

I. INTRODUCTION

Julia is an emerging programming language designed for numerical analysis and data intensive computation. Since its first public release in 2012, Julia has been used successfully in many fields such as data visualization, parallel computing, data science, machine learning, and high-performance computing. In 2017, Julia achieved peak performance of more than 1 Petaflop per second making Julia the fourth language to achieve this goal following C, C++ and Fortran [1]. Due to its advantages such as high performance and programming flexibility, Julia is growing rapidly in recent years. As of January 2022, it has over 35 million downloads [2] with adoptions by over 1,500 universities and 10,000 companies [4] [5] world-wide.

* Corresponding author.

Targeting the high performance and real-time computing application scenarios, one key design goal for Julia is to create an unprecedented combination of ease-of-use, power, and efficiency in a single language. To achieve this goal, Julia reuses legacy high performance libraries in numerical computing developed in C, C++, or Fortran. To this end, Julia introduced a new foreign function interface (FFI) mechanism called `ccall` [11], which supports interactions between Julia and native code with negligible performance penalty. Due to its performance advantages, the `ccall` mechanism has been used extensively in many Julia applications. For example, the prevalent OpenCL [33] library makes use of `ccall` to perform heterogeneous computing, and the Knet [40] machine learning application makes use of `ccall` to perform training.

Unfortunately, for maximum performance, Julia does not provide any necessary security mechanisms and protections against `ccall`. As a result, in multilingual Julia applications, the native code can access the Julia owned memory in arbitrary manners, which may further lead to two serious memory safety issues: the first one is *unmanageable memory accesses*, which denotes that the native code can perform unintended accesses to the memory managed by Julia code. Such accesses include, but not limited to, reading or overwriting the Julia owned memory, which may further lead to sensitive or secrete information leaking or even program crashes. It's challenging to address this issue as the protection mechanism introduced should guarantee as little performance penalty as possible, otherwise, the security protection will defeat Julia's high performance guarantee. Thus, existing studies like sandboxing [18] [21] [23] are not feasible due to their unacceptable performance penalty. For example, the penalty of Vx32 [21] is more than 30% for compute-intensive workloads.

The second security issue is the *unchecked memory access*, which denotes that the native code performs uncontrolled access to the Julia owned memory. For example, native code may write into read-only Julia data structures, or may write beyond the end of Julia buffers without fine-grained access controls. It's challenging to address this security issue, as prior studies like FFI security [9] [13] [28] or control flow integrity (CFI) [17] [25] [26] is coarse-grained in nature, thus infeasible to protect Julia memory at data structure granularity. For example, the minimum granularity libmpk [13] can protect is only one physical page. Therefore, a fine-grained and Julia semantics-aware protection mechanism is needed.

To address the aforementioned challenges, our goal, in this paper, is to design and implement the *first* infrastructure to solve Julia FFI security issues effectively and efficiently. To achieve this goal, we propose a framework dubbed SAFEJ, which consists of two key components: 1) a *privilege separation* by memory protection, to protect Julia memory from unmanageable accesses from unsafe native code; and 2) an *indirection table*, to deal with the unchecked memory accesses to Julia data structures from unsafe native code. The privilege separation is based on the latest Intel Memory Protection Keys (MPK) [15], 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 used by Julia code to store important meta information about Julia data structures, in separate memory page groups protected by MPK. Hence all accesses to these Julia data structures are sanity checked against there indirection table to guarantee only legal ones are allowed.

We have implemented a prototype for SAFEJ, and conducted extensive experiments to evaluate its effectiveness, performance, and usefulness. First, to evaluate the effectiveness of SAFEJ, we applied SAFEJ to micro-benchmarks, and experiments results demonstrated that SAFEJ is effective in protecting Julia memory from malicious native code.

Second, to evaluate the performance of SAFEJ, we applied SAFEJ to 4 large, real-world, widely-used Julia applications: OpenCL (a heterogeneous computing framework), HTTP (a Web server), JuliaDB (a high-performance Database), and Knet (a machine learning library). For OpenCL, the performance overhead SAFEJ introduced is less than 4.7%. For HTTP, the response delay for Web requests is less than 10 nanoseconds. For JuliaDB, the overhead does not exceed 6.4%. And for Knet, the overhead introduced is 2.17%.

Finally, experimental results demonstrate SAFEJ is easy to use. As SAFEJ translates the target Julia applications automatically, thus no developer intervention or manual code rewriting is required.

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

- We conducted a systematic study of Julia FFI security issues.
- We presented an infrastructure dubbed SAFEJ and its prototype, to secure Julia FFI effectively and efficiently.
- We conducted extensive experiments to demonstrate the effectiveness, performance, and usefulness of SAFEJ to large, real-world, and wide-used Julia applications.

The rest of this paper is organized as follows. Section II presents an overview of the background and the threat model for this work. Section III presents the design of SAFEJ and Section IV presents a prototype implementation. Section V presents the experiments we performed, and answers to the research questions based on the experimental results. Section VI discusses directions for future work. Section VII discusses the related work, and Section VIII concludes.

II. Background and Threat Model

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

A. Julia

Julia is a relatively new high-performance programming language designed for numerical analysis and scientific computing. Julia is designed and evolved with three important principles: 1) Julia should be a flexible and dynamic language; 2) Julia should be applicable to scientific computing scenarios which are data intensive; and 3) Julia should be of high performance comparable to traditional statically-typed languages. With these principles, Julia builds upon the lineage of mathematical programming languages, but also borrows much from popular dynamic languages, including Lisp [8], Perl [24], Python [27], Lua [30], and Ruby [46]. To achieve high performance, Julia uses static type inference, without incurring runtime penalty due to dynamic type checking. Moreover, Julia makes use of Just-In-Time (JIT) compilation based on the LLVM compile infrastructure [51]. These designs make the computational efficiency of Julia exceeds that of other dynamic languages, and even rivals that of statically-compiled languages.

The ecosystem of Julia has been growing rapidly since it was first open sourced in 2012. With contributions from thousands of open source developers, the Julia community [34] has more than 7,400 packages registered, including various mathematical libraries, data manipulation tools, and utility packages. These Julia packages cover many application scenarios, such as data visualization and plotting, online computations of streaming data, machine learning, scientific computing, parallel and heterogeneous computing, and distributed computing.

Julia is gaining more popularity and becoming an increasingly important language in recent years. More than 1,500 colleges and universities, such as MIT, Stanford, and UC Berkeley, are using and teaching Julia [4] [5]. More than 10,000 companies around the world use Julia to develop high performance systems, including Google, Intel, Microsoft, Pfizer and NASA [36].

As a result, Julia should be reliable and secure, as it's becoming an important computing infrastructure. Otherwise, tens of thousand companies or users will be affected.

B. Julia FFI

To reuse legacy numerical computing libraries in C/C++ or Fortran, Julia introduced a new language feature `ccall` [11], its foreign function interface mechanism.

Julia FFI design takes a no boilerplate philosophy: native functions can be called directly from Julia without any glue code, code generation, or compilation. To be specific, Julia's 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, allowing native code to invoke Julia functions.

Next, we introduce these two types of interactions separately.

Julia invokes native code. Julia code can invoke functions located 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"), retType, (
    argTypeList, ), argList)
```

where `libPath` stands for the absolute path containing the library `lib`, and `retType`, `argTypeList`, and `argList` are the native function's return type, argument types, and arguments, respectively.

Julia's JIT compiler can generate the same machine instructions for `ccall` as it does for a native call, thus the resulting overhead is the same as calling a library function from C code [22]. Furthermore, by passing pointer arguments into native code, Julia allows the 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 sample code snippet presents a minimal function to execute a piece of Julia code, where a Julia command is executed by calling a specific Julia API `jl_eval_string()`.

```
int main(int argc, char *argv[]){
  jl_init(); /* setup the Julia context */
  jl_eval_string("println(sqrt(2.0))"); /* run
      Julia commands */
  jl_atexit_hook(0); /* notify Julia that the
      program is about to terminate */
  return 0;
}
```

This set of APIs, which support various functionalities such as data types conversion, memory management, and exception handling, makes it easy for users to integrate Julia code into larger 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`. The key advantage of MPK over existing technology such as page table protection is that MPK doesn't need to switch to kernel space to manipulate the page table and translation lookaside buffer (TLB), thus is more efficient.

Many software abstractions, such as ERIM [10] and libmpk [13], have been proposed to make it easier to use the Intel MPK hardware technology. For example, the relatively new libmpk abstraction provides a group of APIs supporting page permission switch, memory pages allocation, initialization and free, etc. 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; internally, libmpk maintains a mapping from virtual keys to page groups, to avoid scanning all pages at this destruction step. 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.

Intel MPK, along with these software abstractions, shows significant performance advantages. For example, libmpk is used to protect some real-world applications: OpenSSL, JavaScript JIT compiler, and Memcached. Experimental results showed that the runtime overhead libmpk introduced is less than 1% at a high frequency of switching permission, while provides over an x8 performance improvement over the traditional `mprotect` system call for process-level permission switch [13].

D. Threat Model

The focus of this work is on security issues arising when Julia interacts with native insecure code such as C/C++ in multilingual Julia applications. Therefore, we make the following assumptions in the threat model for this work.

We assume that the host environment, running the Julia applications, has standard protections. For example, the underlying hardware or operating systems provide standard protections such as Data Execution Prevention (DEP) [74], Stack Canaries [75], and Address Space Layout Randomization (ASLR) [76]. Furthermore, the Julia compiler has not been compromised by malicious attackers so that the binaries generated from the compiler are trustworthy. It should be noted that although operating systems and compilers security studies are very important, they are independent of and orthogonal to the study in this work; and these research fields can also benefit from the research progress in this work.

We assume that pure Julia code is safe and will not pose a security threat to the application being investigated. For example, pure Julia 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 native code Julia interacting with is untrusted and unreliable. For example, if the native function being called through Julia FFI is vulnerable, attackers can control the native code to perform arbitrary attacks such as illegal memory read or write, or triggering buffer overflows. As our focus in this work is to study the Julia FFI security,

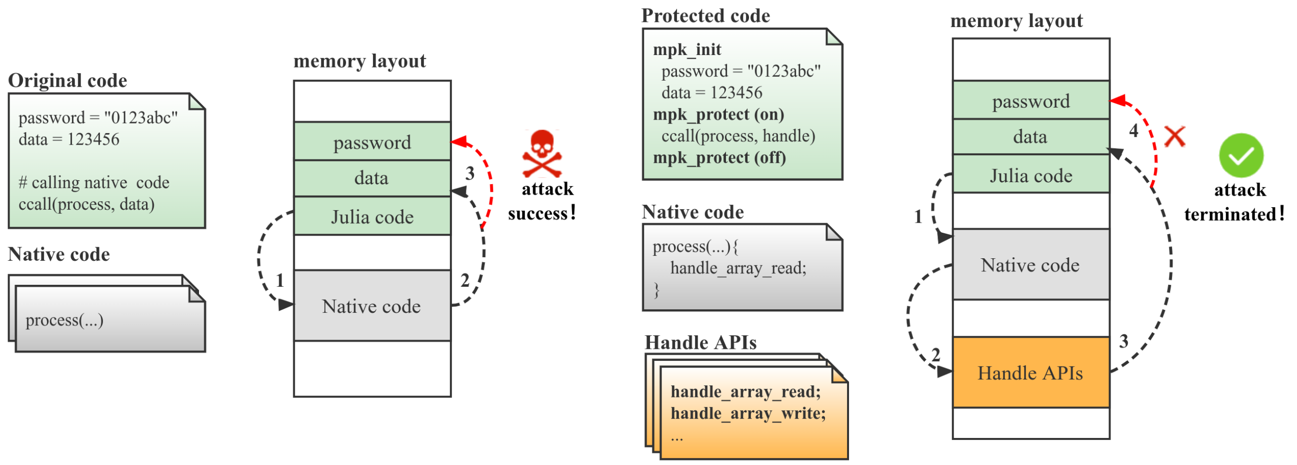


Fig. 1: The overall design of SAFEJ: 1). privilege separation: when the control transfers from Julia to native code, access privileges are turned off for dedicated Julia memory with lock functions; 2). pointer sanitizing: Julia memory is only allowed to be accessed by trusted handle APIs which manage fine-grained accesses.

so the results in this work can be used to strengthen classical native code protection such as control-flow integrity (CFI) [17] [25] [26].

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, such as ERIM or libmpk, is secure. Although testing these abstractions or implementations are important research topic, it's irrelevant to and out of the scope of this work.

III. SAFEJ DESIGN

In this section, we present the framework of SAFEJ by describing its design in detail. We first discuss the design principles (Section III-A), then present privilege separation by heap isolation (Section III-B). Finally, we present the design of the indirection table to sanitize pointers (Section III-C).

A. *Overall Design*

Fig. 1 presents the overall design for SAFEJ. We have two important principles guiding the design of SAFEJ: first, SAFEJ should be flexible enough to protect different interactions between Julia and native code as control transfers. As shown in Table I, there are four possible control transfers in a multilingual Julia application: 1) control transfers between Julia code, as the threat model of this work (see Section II-D) specifies that Julia code is trusted, so no special protection technologies are required for this scenario; 2) control transfers from Julia code to native code through Julia FFI, as the the native code are not trusted, SAFEJ utilizes privilege isolation to isolate the Julia memory to ensure that native code cannot perform unmanageable accesses; 3) control transfers from native code to Julia, SAFEJ utilizes an indirection table to guarantee that native code can only access Julia's memory in a fine-grained, controlled and secure manner; and 4) control transfers between native code, such kind of transfers are out of the scope of this work and standard protection such stack canary or CFI can be enforced. It's important to note that, although we discuss these protection technologies separately for ease-of-presentation and understanding, they are really a unified approach in guaranteeing Julia memory safety.

TABLE I: Protection Technologies for Different Interactions

Control Transfer	Description	Protection
Julia $\rightarrow$ Julia	Julia code accessing Julia-allocated memory	/
Julia $\rightarrow$ Native	Julia code accessing native code allocated memory	Privilege separation
Native $\rightarrow$ Julia	Native code accessing Julia-allocated memory	Indirection table
Native $\rightarrow$ Native	Native code accessing native code allocated memory	/

The second principle is that SAFEJ should be fully automated thus minimal interventions are required from developers. SAFEJ is proposed to be used in two scenarios: 1) new Julia projects development; and 2) legacy Julia projects migration. In the first scenario, developers can integrate SAFEJ in their projects without difficulty, as SAFEJ provides an effective programming model and a group of easy-to-use APIs. In the second scenario, SAFEJ provide automated tools to help developers migrate their legacy code, which will be further discussed next.

B. *Privilege Separation by Memory Isolation*

In this section, we present the design of page groups and privilege separation, respectively.

1) Page Groups: Intel MPK enforces memory protection based on *page groups*, all pages in one page group share the same protection key thus have the same privilege permissions.

According to the Intel manual, newest line of Intel CPUs support up to 16 different page groups physically [14]. 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.

As SAFEJ is designed to isolate memories of Julia and native code, it splits the memory into two disjoint page groups: the Julia memory and the native memory, as described by Fig. 1 (the green region and the gray region, respectively). SAFEJ 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's possible to make use of more protection keys (thus more page groups), two is enough in this work for SAFEJ to secure Julia FFIs; and 2) pages in one page group need not to be adjacent but may interleave. For example, a Julia memory page sits between two native pages, which makes this protection strategy more flexible and convenient.

2) 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. SAFEJ is designed to make use of two distinct groups of permissions to protect the two memory regions (the Julia memory and the native memory) respectively. Julia code has full access permissions to the two memory regions, whereas the native code only has permissions to access the native memory.

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

It's important to note that since the MPK API is a user-level protection mechanism, that is, the `rdpkru` and `wrpkru` MPK instructions are non-privileged, so in theory, the native code can also makes use of these instructions to switch Julia memory permissions, which breaks the security guarantees Intel MPK enforced. Thus, one must ensure that native code doesn't switch the permissions of the Julia memory. To achieve this goal, one can leverage any standard binary scanning techniques [29][31], 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 Section VI.

C. Pointer Sanitizing by the Indirection Table

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

Technically, this technology consists of four components: 1) the handle, 2) the indirection table, 3) the handle API, and 4) external function conversions, which will be discussed next, respectively.

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

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

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

Second, keys in the indirection table (that is, handles) must be random enough thus be difficult to guess or forgery. Otherwise, suppose that an adversary can guess a valid handle which does not belong to her, then by looking up the indirection table with that guessed handle, she is able to access a Julia memory address she had no permissions.

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

3) Handle API: to allow external functions to make easy use of handles, SAFEJ designed a group of *handle APIs*, which are implemented in Julia and exposed to native code. The handle APIs should include common operations on Julia, such as data structures manipulation, memory management, and exception handling.

It's important to note that handle APIs are supplements and enhancements to the standard Julia C APIs. The standard Julia C APIs can be classified into two categories: 1) APIs that don't access Julia memory, these APIs can continue to be used by native code as it's secure; and 2) APIs accessing Julia memory, these should be replaced by a corresponding handle API which is secure.

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

Although manual converting the native code is possible, doing so is laborious and error-prone, especially for large Julia projects with considerable native code bases. To this end, an automatic technology is required to achieve this in practice, which we will discuss further in the next section.

IV. SAFEJ IMPLEMENTATION

In this section, we present a prototype implementation for SAFEJ. We first introduce the implementation of privilege separation (Section IV-A), then the implementation of indirection table (Section IV-B).

A. Implementation of Privilege Separation

SAFEJ leverages the libmpk library 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. SAFEJ implements two page groups, that is `GROUP_JULIA` and `GROUP_NATIVE`, for the Julia memory and native memory, respectively. These two page groups are distinct integers to be used by libmpk functions. It's interesting to note that although the Intel CPU reserves 4 bits to represent the page groups (the 32nd to 35th bits in the page table entry), which indicates that the valid page groups are in the range $[0, 16)$, libmpk supplies infinite number of page groups by virtualizing the physical ones, which greatly simplifies the application implementation.

SAFEJ utilizes the `mpk_mmap()` function, to allocate memory in corresponding page groups. For example, SAFEJ 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. SAFEJ initialized read/write permissions for both the Julia memory and native memory, so Julia code has full access to both memories initially. To enforce privilege separation, as Fig. 1 shows, SAFEJ disables the permissions of the Julia memory before each `ccall`, by calling `mpk_protect(GROUP_JULIA, PROT_NONE)`. Thus, when control transfers to native code, native code has no access to the Julia memory, which guarantees memory safety. When control transfers back from native to Julia, SAFEJ restores permissions by calling `mpk_protect(GROUP_JULIA, PROT_READ|PROT_WRITE)`.

The library libmpk suggested a pattern to leverage `mpk_begin()` and `libmpk_end()` APIs to setup and clear permissions. 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 the prototype for SAFEJ, that this pattern is not applicable to SAFEJ, as the permissions are being disabled during the `ccall`, instead of being enabled.

B. Implementation of Indirection Table

SAFEJ makes use of the indirection table to sanitize pointers from native code to Julia.

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

It should be noted that although using more bits, such as 128 bits, to represent handles will lead to to higher randomness, it makes passing handles to native 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, the SAFEJ prototype makes use of hash table data structures to implement a new datatype `itable` for the indirection table. Furthermore, SAFEJ's current implementation supports the following APIs as shown in TABLE II. These functions have similar semantics to standard APIs in most 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.

TABLE II: The 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

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

Implementation of Handle APIs. To allow external functions to access Julia memory securely through handles, current SafeJ prototype implements a group of handle APIs. These APIs provide read/write capabilities for different Julia data structures: getter/setter APIs for fields in a structure; read/write operations for arrays, and so on. SafeJ implements these APIs in Julia and exposes them to native code. For example, the following sample code snippet

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

implements a handle API `handle_array_read`, for reading an element located at index `index` from the array represented by the argument `handle`. Error recovery code is omitted for simplicity. When the native code needs to access an array element represented by `handle`, it transfers control to Julia by invoking the above function. The Julia code looks up the indirection table `itable` to search the address `arr_addr` for the array, then returns the array element at `index` by transferring control back to native code.

External Function Conversion. Handles are used differently in two scenarios: in new Julia projects or in legacy ones. For the former, it's straightforward for Julia developers to integrate the new handle APIs to develop native code. The latter scenario is much more challenging, as the existing legacy code must be converted to use the new handle APIs.

Fig. 2 presents a simple yet illuminating native code sample.

```
// before conversion
int native_read_array(int *array, int index){
  return array[index];
}
// after conversion
int native_read_array(long long handle, int
    index){
  jl_value_t *result = jl_call2(
      handle_array_read, handle, jl_box_int32(
      index));
  int elem = jl_unbox_int32(result);
  return elem;
}
```

Fig. 2: Before and after a native function is converted.

The original native code is vulnerable as it reads from a raw address `array`, which may lead to serious memory safety issues such as buffer overflows. Two major modifications are required to convert this native code: 1) the function argument is converted from a raw address `array` to a `handle` (recall that SafeJ makes use of 64 bits integers to represent handles); and 2) the direct array access `array[index]` is converted into invocations to the corresponding handle APIs.The function argument `index` and return value `result` are also boxed or unboxed, respectively, to coordinate with the Julia memory model.

To mitigate this, SafeJ developed a prototype translator to convert the native code automatically. This translator is based on the CIL infrastructure [82] and works in three major steps: 1) the native sources are parsed into abstract syntax trees (AST), a compiler internal data structure suitable for manipulation; 2) SafeJ automatically rewrites the target function to convert its argument and body as discussed above; this rewriting is essentially a syntax-directed AST conversion and is quite efficient in practice; and 3) SafeJ generates converted native code by outputting the rewritten AST.

V. Evaluation

In this section, we present experiments to evaluate SafeJ. We first present the research questions guiding the experiments (Section V-A), then the effectiveness of SafeJ on micro-benchmarks (Section V-C). Next, we evaluated the performance and usefulness of SafeJ (Section V-D) on large and real-world Julia applications including heterogeneous computing, Web servers, DataBase, and machine learning.

A. Research Questions

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

RQ1: Effectiveness. As SafeJ is proposed to secure Julia FFIs, is it effective in protecting Julia programs against malicious native code?

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

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

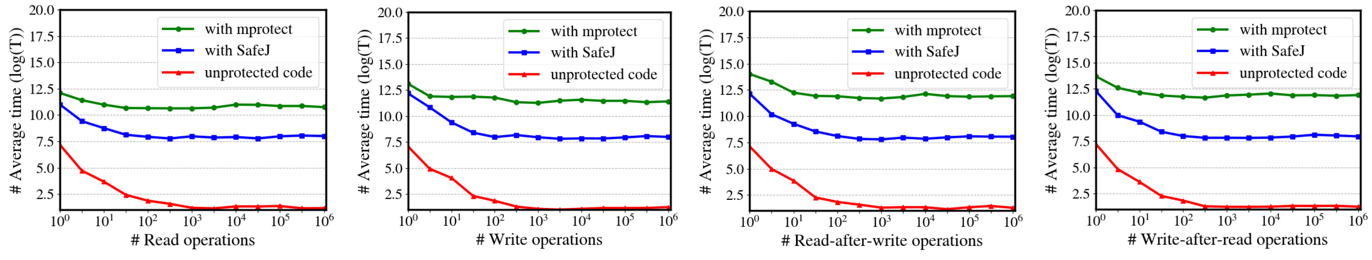
B. Experimental Setup

All the experiments and measurements are performed on a server with one 4 physical Intel i7 core (8 hyperthread) CPU and 16 GB of RAM running Ubuntu 20.04.

C. Effectiveness

To answer **RQ1** by demonstrating the effectiveness of SafeJ, we conducted experiments to testify the two protection mechanisms: privilege separation and indirection table, respectively.

First, we construct a group of micro-benchmarks consisting of multilingual Julia applications, and manually inject common kinds of vulnerabilities into the native code of these applications. For example, we inject arbitrary memory accesses, that



(a) Single Read (b) Single Write (c) Read-after-Write (d) Write-after-Read

Fig. 3: Privilege separation performance evaluation results for four scenarios: read, write, read-after-write, and write-after-write.

is, by casting an arbitrary integer into a pointer, the native 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 native 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 SAFEJ to these benchmarks, compiled and executed them for a second time, we observed SAFEJ 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.

These results demonstrated that SAFEJ is effective in protecting multilingual Julia applications from common memory attacks.

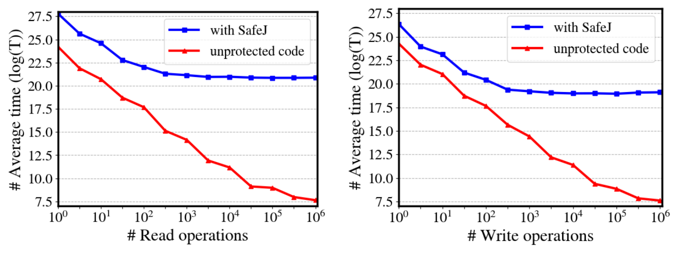
D. Performance

To answer **RQ2** by investigating the performance and overhead SAFEJ introduced, we testified micro benchmarks for four memory access scenarios: 1) read-only; 2) write-only; 3) write-after-read; and 4) read-after-write. 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 (red); 2) the one with SAFEJ (blue); and 3) the one with `mprotect` (green). 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 $y = \log(T)$, where T is the original absolute running time.

We observed that, for the four scenarios, the average overhead SAFEJ 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 1600 and 2800 nanoseconds, respectively. And for read-after-write and write-after-read operations, the overhead are 4600 and 4100 nanoseconds, respectively.

These experiments results demonstrated that SAFEJ is efficient with negligible overhead.



(a) Single Read (b) Single Write

Fig. 4: Indirection table performance evaluation results for two scenarios: single-read and single-write.

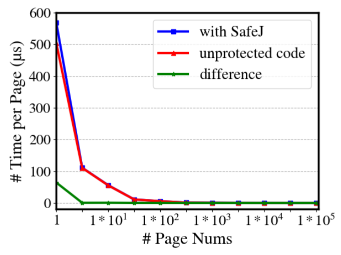
Indirection Table Performance. To investigate the overhead introduced by the indirection table, we developed a multilingual Julia benchmark which access arrays located in Julia memory. With this baseline, we created a safe version of this benchmark by replacing all pointer parameters by handles, as well 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 numbers of operation, and y-axis presents the average running time (again in logarithmic time).

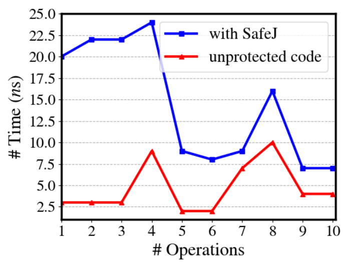
For read and write operations, the indirection table adds 7100 and 23000 nanoseconds on an average, respectively. This overhead is practical and acceptable for two reasons: 1) it's in par with prior studies on runtime sanitizers; and 2) this performance penalty only exists on those native functions that invoke handle APIs, while others don't involve this "pay-as-you-go" penalty.

E. Usefulness

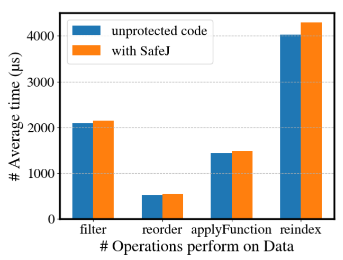
To answer **RQ3** by demonstrating the usefulness of SAFEJ, we applied it to four large and real-world Julia applications from 4 representative fields: heterogeneous computing, Web servers, Database, and machine learning. It should be noted



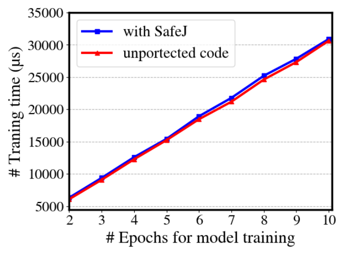
(a) OpenCL: Average time spent for per memory page.The difference indicates the average overhead introduced.



(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 data.



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

Fig. 5: SAFEJ performance evaluation results for OpenCL (heterogeneous computing), HTTP (Web server), JuliaDB (DataBase), and Knet (machine learning).

that the effectiveness of SAFEJ was verified on these four applications by injecting memory attacks intentionally. Thus, we focus on its performance penalty in this section.

1) Heterogeneous Computing: **Heterogeneous Computing:** OpenCL [33] is a popular package which aims to offer a complete solution for heterogeneous computing in Julia. A Julia application incorporating OpenCL will invoke, via `ccall`, specific native functions to complete some core functionalities such as context construction, events triggering, and computation initiation.

We modified OpenCL by introducing SAFEJ and recompiled it to obtain a safe version `openCL.safe` to compare it with the original version `openCL.orig`. 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 decreases. And the overhead SAFEJ introduced is less than 4.7% on average.

2) **Web Servers:** HTTP [35] is an open source and prevalent software package for developing client or server web applications in Julia. In HTTP, Julia make extensive use of `ccall`, to invoke native functions. These native functions perform critical tasks such as initializing sockets, manipulating configuration strings, reusing server ports.

We compiled the HTTP without and with SAFEJ to two binaries `http.safe` and `http.orig`, respectively. We then deployed two Web servers responding GET requests from clients. We then recorded the average responding time of the two Web servers for 10 rounds, each with 10000 requests.

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:** JuliaDB [37] is a widely-used Database in Julia [38]. JuliaDB invokes, through `ccall`, native functions to perform key operations such as data loading, and reading parameters from the memory pool.

We applied SAFEJ to JuliaDB and compiled the source code to obtain two binaries, `juliaDB.safe` and the original `juliaDB.orig`. Then, we ran the two Database binaries on Hflights.csv [39] data source. In our 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 SAFEJ for the four operations are 2.86%, 3.80%, 6.39%, and 2.73%, respectively.

4) **Machine Learning:** Knet [40] is a popular deep learning framework implemented in Julia, and ranks second in all Julia AI packages [41]. In model training phase, Knet invokes, through `ccall`, native functions to perform tensor operations such as reduction, broadcast, deepcopy, and deepmap.

We revised and recompiled Knet to obtained two versions of the library: `Knet.safe` and `Knet.orig`. Then, we wrote 2 digit recognition applications with MNIST [42] based on LeNet [43], and used SAFEJ 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 SAFEJ is 2.17% on average. Furthermore, SAFEJ does not affect the accuracy of the model, and training data were not modified.

Experimental results in this section demonstrated that SAFEJ is useful to protect real-world large Julia applications.

VI. 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 understanding Julia FFI security issues, and proposing an effective protection technology for mitigation.

Binary analysis. Intel MPK is very efficient, as it allows processes to switch permissions at user space, with two non-privileged MPK instructions `rdpkru` and `wrpkru`. A consequence of this feature is that a malicious attacker can abuse these instructions to switch permissions. In this work, we have leverage static binary analysis to scan the native code for these instructions, in par with existing studies. Once detected, SAFEJ will report to the users. Although we don't encounter in our experiments, some native code, such as Just-In-Time compilers (JITs) [77][78] or Self-Modifying-Code (SMCs) [79][80][81], may generate code at runtime, which poses challenges for detection. For such cases, we believe one promising approach is to perform dynamic code analysis [53][54][55][56].

Error injection. To evaluate the effectiveness of SAFEJ, we manually injected specific memory errors into the native code of target multilingual Julia applications. The experimental results demonstrated that this approach is effective in performing the evaluations on micro-benchmarks. To conduct experiments on Julia applications in the wild, an automatic method is desired. To this end, one promising approach is to make use of automatic error injection tools [57][58], which can minimize the manual effort required.

Other hardwares. Intel MPK is a relatively new memory protection technology, which was used in this paper to protect Julia FFIs. Similar to Intel MPK, other hardwares mechanisms, such as IBM Storage Protection [44] and ARM Domains [45], provide memory key protection as well. We believe the technology presented in this work can also be used on these hardwares due to the similarity between hardware features, and we leave the extension of SAFEJ to other hardwares a future work.

VII. Related Work

In recent years, there are a significant amount of studies on security of native code, FFI, and memory protections. However, the work in this paper stands for a novel contribution to this field.

Native Code Security. There are a lot of studies on native code security. With a combination of static and dynamic checks, George C et al. [47] presented CCured which guarantees type safety for legacy C programs while retaining the syntax and semantics of C. Trevor Jim et al. [48] proposed Cyclone to prevent the buffer overflows, format string attacks, and memory management errors in C programs. Wang et al. [49] proposed a polymorphic SSP (P-SSP) technology to re-randomize the canaries new process to increase the difficulty for attackers to get the canaries. Jang et al. [50] propose a technology of code replacement to prevent buffer overflow. Ren [59] 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 Julia FFIs. Alternatively, these studies can also be deployed along with SAFEJ, to secure the Julia interaction with native code.

Foreign Function Interface Security. The FFI mechanism, in many languages, has been a major attacked surface, thus gaining a lot of attention. Rivera et al. [28] proposed a framework called Galled that secured Rust FFIs. David Terei et al. [71][72] presented Safe Haskell which securely executes arbitrary unsafe code with no overhead. Furr and Foster [60] proposed a multi-language type inference system to check OCaml external function interface calls, in order to prevent memory safety violations. They further extended the system to check the type safety of Java Native Interface (JNI) programs [61]. Tan and Li et al. [62] proposed the SafeJNI framework to guarantee type safety when calling native C methods, by performing static and dynamic checks on C code. Tan and Li [63][64][65] built a new static analysis framework for the difference between Java and native method exception handling methods to find exception handling errors in Java native code. And on the basis of these work, further conducted empirical security research [66] on native code in JDK, and put forward some vulnerability modes. The Python/C API security-related research is rare. Pungi [67] uses affine abstraction to statically analyze the SSA form of the program. A key limitation of existing studies is that there is no systematic investigation on the Julia FFI security issues and mitigations. On the contrary, this work, for the first time, investigates and systemizes Julia FFI security issues and their mitigations.

Hardware Features for Memory Protection. The are many research progress on proposing new hardware primitives to protect memory. Intel proposes SGX [68] instruction set extension, which implements enclave container with hardware technology to provide confidentiality and integrity protection for code and data. Intel MPK [15] added specific register and instructions to reduce the overhead of switching page access permissions. Similar to the concept of Intel MPK, IBM proposed the Storage Protection primitive [44], and on the ARM platform there is ARM Domains [45]. Shreds [69], ARMlock [70] and FlexDroid [73] isolate insecure code or third-party libraries from sensitive data for ARM platforms. While the focus of this work is not to invent new protection primitives, but to enhance Julia FFI security by leveraging the latest existing hardware protection features.

VIII. Conclusion

This paper presented SAFEJ, an infrastructure to secure Julia FFIs, which consists of two key components: privilege separation for protecting Julia memory from native code access, based on the latest Intel MPK technology; and an indirection table to sanitize pointers from native code. We implemented a prototype for SAFEJ and conducted systematic experiments with it. The experiment results demonstrated the effectiveness, efficiency, and usefulness of SAFEJ. Overall, the work in this paper is a first step towards securing the Julia FFIs, making Julia not only an efficient but also a safer programming language.

References

[1] Regier J, Fischer K, Pamnany K, et al. Cataloging the visible universe through Bayesian inference in Julia at petascale[J]. Journal of Parallel and Distributed Computing, 2019, 127: 89-104.

[2] https://juliacomputing.com/blog/2020/08/14/newsletter-aug.html

[3] https://spectrum.ieee.org/top-programming-languages/

[4] https://juliacomputing.com/media/2022/02/julia-turns-ten-years-old/

[5] https://juliacomputing.com/blog/2022/01/newsletter-january/

[6] https://juliacomputing.com/case-studies/astra-zeneca/

[7] https://juliacomputing.com/case-studies/pfizer/

[8] https://lisp-lang.org/

[9] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[10] 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).

[11] https://docs.julialang.org/en/v1/base/c/#ccall

[12] https://github.com/JuliaGPU/OpenCL.jl

[13] 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).

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

[15] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, May 2019.

[16] https://julialang.org/

[17] 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.

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

[19] https://docs.julialang.org/en/v1/manual/embedding/

[20] https://github.com/JuliaLang/julia

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

[22] https://docs.julialang.org/en/v1/manual/embedding/

[23] 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.

[24] https://www.perl.org/

[25] 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.

[26] 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.

[27] https://www.python.org/

[28] Elijah Rivera, Samuel Mergendahl, Howard Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. Association for Computing Machinery, New York, NY, USA, 824–836. DOI:https://doi.org/10.1145/3485832.3485903

[29] 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).

[30] https://www.lua.org/

[31] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight, User-level Sandboxing on the x86. In USENIX Annual Technical Conference.

[32] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In 2019 IEEE Symposium on Security and Privacy (SP).

[33] https://github.com/JuliaGPU/OpenCL.jl

[34] https://juliahub.com/ui/Packages

[35] https://juliahub.com/ui/Packages/HTTP/zXWya/0.9.17

[36] https://www.infoq.cn/article/b0dpmasunf3lbb8y2svq

[37] https://juliadb.org/

[38] https://juliapackages.com/packages?search=database

[39] https://github.com/selva86/datasets/blob/master/hflights.csv

[40] https://github.com/denizyuret/Knet.jl

[41] https://juliapackages.com/c/ai?sort=stars

[42] http://yann.lecun.com/exdb/mnist/

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

[44] IBM. Power ISATM Version 3.0 B, 2017.

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

[46] https://www.ruby-lang.org/en/

[47] 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.

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

[49] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, Bing Mao: To Detect Stack Buffer Overflow with Polymorphic Canaries. DSN 2018: 243-254.

[50] 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)

[51] https://llvm.org/

[52] https://juliacomputing.com/case-studies/mit-robotics/

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

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

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] Jiadong Ren, Zhangqi Zheng, Qian Liu, Zhiyao Wei, Huaizhi Yan: A Buffer Overflow Prediction Approach Based on Software Metrics and Machine Learning. Secur. Commun. Networks 2019: 8391425:1- 8391425:13 (2019).

[60] Michael Furr, Jeffrey S. Foster: Checking type safety of foreign function calls. PLDI 2005: 62-72

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

[62] Gang Tan, Andrew W. Appel, Srimat Chakradhar, Anand Raghunathan, Srivaths Ravi, Daniel Wang: Safe Java Native Interface. IEEE International Symposium on Secure Software Engineering, 2006: 97¨C106.

[63] Siliang Li, Gang Tan: Finding bugs in exceptional situations of JNI programs. CCS 2009: 442-452.

[64] Siliang Li, Gang Tan: JET: exception checking in the Java native interface. OOPSLA 2011: 345-358.

[65] Siliang Li, Gang Tan: Exception analysis in the Java Native Interface. Sci. Comput. Program. 89: 273-297 (2014).

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

[67] Junjie Mao, Yu Chen, Qixue Xiao, Yuanchun Shi. RID: Finding Reference Count Bugs with Inconsistent Path Pair Checking. ASPLOS 2016: 531-544.

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

[69] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained Execution Units with Private Memory. In Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, May 2016.

[70] 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.

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

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

[73] 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.

[74] Microsoft. 2006. A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. Online. http://support.microsoft.com/kb/875352/en-us

[75] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. 1999. Protecting systems from stack smashing attacks with StackGuard. In Linux Expo.

[76] PaX. 2003. PaX Address Space Layout Randomization.

[77] 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.

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

[79] 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.

[80] 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.

[81] 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.

[82] 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.