SGXPy: Protecting integrity of Python applications with Intel SGX

Denghui Zhang, Guosai Wang, Wei Xu, Kevin Gao
Institute of Interdisciplinary Information Sciences
Tsinghua University
Beijing, P.R.China
{denghuizhang, wgs14, weixu}@tsinghua.edu.cn, kevingao96@gmail.com

***Abstract*—Python is the programming language of choice for many data scientists, and thus widely used in cloud computing platforms. Untrusted cloud environments have imposed challenges to the security of Python applications. Intel SGX (Intel Software Guard eXtensions) provides an encrypted enclave for securing applications, and a library OS technology can be adopted to run legacy applications inside these enclaves. However, this technology has some limitations: (`i`) It is difficult to ensure the integrity of Python applications as a result of the complex dependencies among modules. (`ii`) Python applications often spawn new processes, and file access permissions need to be handled separately in the parent-child process. To address these limitations, we present SGXPy (SGX Python), an integrity preserving tool for Python applications. The design of SGXPy makes it possible to obtain dependencies of applications and assign file access permissions among processes automatically: (`i`) During the build stage, SGXPy constructs dependency manifests of Python applications based on the `ptrace` mechanism. (`ii`) To enhance access control among processes, SGXPy utilizes process introspection to cascading manifests for each process. With the proposed framework, sophisticated Python applications such as NumPy and a web server can now run unmodified with the library OS. We present a series of experiments to evaluate performance overheads of Python applications in SGX. Our evaluation of NumPy submodules shows SGXPy can pass 97.60% of unit testing, even with the isolated environment and limited memory of SGX.**

***Keywords*—SGX, Python, trusted computing, integrity**

I. INTRODUCTION

As the most popular language in 2018[1], Python is widely used in data science. Data scientists have developed robust ecosystems of libraries to help them perform analytical work [1], [2]. These libraries shorten the amount of time it takes for scientists to go from project inception to meaningful work.

Large scale data processing relies on cloud computing for high scalability and low cost. One of the main challenges in cloud computing is protecting the integrity of the outsourced data and code. When running Python applications in the remote and potentially adversarial environment, Intel SGX can ensure that the sensitive data and code processed in an enclave are not disclosed or modified by the host OS, hypervisor, and any other higher priority software.

The SGX enclave is designed to run in user-space, and therefore relies on the operating system to access I/O resources. If an unmodified application requests disallowed system calls inside of an enclave, shielding code in a library OS [3], [4] can sanitize system call results before returning it to inside of the enclave.

The library OS requires a manifest of an application to specify which resources are allowed to use in runtime. The integrity of a runtime environment can be validated by verifying hash values for all of dependency files in the manifest.

Current library OS technologies focus on applications developed in the C/C++ programming languages [5], [6]. Third-party modules support for dynamic languages, including Python is still restricted in the library OS. The following reasons explain why:

1) Dynamic languages often rely on a large number of third-party modules for rapid development. It is impractical to specify all of the dependency files manually for a single Python application. While most of the APIs used by applications developed with C/C++ is concentrated in several `.so` files and can be assigned easily.
2) Python often obtains results through invoking existing shell tools rather than direct function calls. The read/write permissions of files are different among the Python main process and forked subprocesses. The authentication information of files in the parent process should not be shared directly to the isolated child processes.

To overcome these barriers, we propose SGXPy, an integrity preserving tool for Python applications. During the build stage of an application, SGXPy automatically extracts its dependency information into a manifest based on the `ptrace` [7] mechanism. During runtime, the library OS loads and verifies the execution environment of the application according to the manifest.

The `ptrace` allows us to intercept all system calls and signals. By leveraging the rich parameter contexts collected from the `ptrace` introspection, SGXPy can build dependencies for a variety of Python applications (both open source or closed source ones), making it not only more compatible, but also more accurate, than other static or dynamic analysis methods.

The contributions of this paper are:

1) SGXPy, a framework utilizing Intel SGX to protect the integrity of Python applications.
2) An automatic manifest construction mechanism that enables programs that can run in SGX to evolve from plain

[1]https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018

2640-0715/19/$31.00 ©2019 IEEE
DOI 10.1109/APSEC48747.2019.00063

applications to sophisticate applications such as NumPy and the web server.

3) An accurate access control method for Python multiprocessing applications. By hooking process involved system calls using the `ptrace`, the method eliminates mismatch of file access permissions among parent and child processes.

The rest of the paper is divided as follows: in Section II, we provide more background on the paper. In Section III, we review related works. In Section IV, We introduce SGXPy. In Section V, we present a series of evaluation based on our proposed solution. Finally, in Section VI, we conclude the paper and discuss some possible work to improve our tool.

II. Background

Python languages are often used in cloud environments to process large chunks of data faster. These untrusted cloud environments have imposed new challenges to the protection of sensitive data and code [8]. In such environments, multiple applications from different owners may reside in the same physical server, making it possible for malicious users to exploit potential vulnerabilities to steal data or compromise applications. Securing data and code to guarantee its privacy and integrity is highly desired by end users. However, software-based security is often insufficient due to vulnerabilities in applications [9].

A. Intel SGX

Intel SGX is a set of off-the-shelf processor extension instructions. It allows user-level code to be executed inside of a protected enclave. SGX can guarantee the confidentiality and integrity of applications within Intel hardware even in the presence of privileged malware.

Different from conventional sandboxes, such as internet browsers or virtual machines, SGX adopts an inverse sandbox model to protect software. When loading code into an enclave, SGX measures the memory areas of code and data blocks to obtain the *measurement result* (MR) value. This calculated *MR* is compared to the MRENCLAVE value contained in the signed application. Only if they match, SGX begins to execute the code. This gives a remote party the confidence that the intended software is running securely within an enclave on an SGX enabled platform.

SGX employs a threat model where it only trusts Intel CPU and the code running in an enclave. The small TCB (Trusted Computing Base) of SGX requires an application to be refactored into two parts: untrusted code and a trusted enclave that the untrusted code can securely call into. It is the user's responsibility to ensure the security of the untrusted code. This mechanism can resist sophisticated attackers who exploit vulnerabilities in privileged software. However, SGX is not bulletproof and is vulnerable to side-channel [10], [11] attacks, which is beyond the scope of this paper.

The trust-separated development model brings challenges to the porting of existing software. For instance, it takes a lot of time to partition code into trusted and untrusted parts. Additionally, there is no guarantee that the reconstructed system is fully compatible with the original interface, which introduces the risk of breaking existing system [12]. One way to run unmodified applications in SGX is to introduce a port of operating system in the enclave. A library OS approach is the implementation of the APIs of OS. It makes legacy applications work out-of-the-box in SGX by emulating arcane system call semantics.

Another challenge for running applications in SGX is the limited memory space. Enclave pages and SGX structures are backed by a range of regular DRAM, which is called EPC (Enclave Page Cache). The size of an EPC is limited to 128MB on version 1 of SGX, only 91 MB of which is usable for an application running in SGX. If the required memory is larger than that available, the data must be encrypted before changing from EPC to DRAM, then decrypted before swapping back into EPC from DRAM [13].

B. Library OS

A library OS corresponds to the minimum set of OS libraries required to run an application. We can execute unmodified applications in SGX by introducing a port of an operating system, or library OS, in the enclave.

Instead of just verifying the application with the conventional SGX programming model, the library OS can protect the integrity of the entire Python runtime environment by checking its dependent resources files.

Dynamic languages often fork child processes to pick up system information. To support multiprocessing in SGX, we can treat each process as an enclave. When a new process is established, a corresponding Enclave B is launched. Then, according to passed arguments, Enclave B launches the intended new process in another library OS.

When forking child processes, the library OS not only starts a new process, but also copies resources such as virtual address space, the file descriptors table, and the signal handlers table from the parent process. We have to authenticate another enclave before sharing these resources, as the new child processes is established by the untrustworthy operating system.

We can use the SGX-provided local attestation mechanism [14] for multiple enclaves running on the same host to validate each other. After the attestation is passed, the parent enclave shares resources to the new process using the symmetric key generated during authentication. The library OS Graphene-SGX adopts this approach [6].

C. Dependency Analysis

To keep the integrity of an application, we must ensure the integrity of all dependency files, loadable modules, etc. before opening these files. We can collect evidence of dependency files during build time. Then, during runtime, the library OS can validate the measurement information of the application and dependency files.

A Python application often imports a large number of modules for reuse and rapid development. These modules recursively rely on other modules. When dependency levels

TABLE I
DEPENDENCY COUNT FROM DIFFERENT TOOLS

script	SGXPy	sfood	modulefinder	PyInstaller
helloworld.py	63	0	0	212
numpytest.py	91	421	717	1225

become deeper, it is impractical to assign all dependency files manually.

We can find the dependency of the program automatically by statically analyzing its AST (Abstract Syntax Trees). However, there are two fundamental limitations to this method, which can be explained by the following two concrete examples. (i) As shown in Listing 1, when no `import` statement is used in the `helloworld.py` script, static analysis method cannot find modules loaded at runtime or built-in modules such as `os` and `sys`. As shown in Table I, the number of dependent files analyzed by AST is 0, while the exact number of dependency files is 63. (ii) An AST analysis may introduce a large number of redundant modules. As shown in Table I, the number of dependent files needed by the `numpytest.py` script is 93, whereas the analysis results of `sfood`[2], `modulefinder` and `PyInstaller`[3] are 421, 717, and 1225 respective modules. Although the `PyInstaller` tool can find dependent files in the two cases, it finds more than ten times as many files as SGXPy. The redundant modules found will be explosive growth for complex applications.

Listing 1. helloworld.py

```
1   #!/usr/bin/env python
2
3   print "Hello, World."
```

Listing 2. numpytest.py

```
1   #!/usr/bin/env python
2
3   import numpy as np
4   print np.arange(15)
```

Redundant files are mainly a result of Python's cross-platform feature. To run code on different platforms, the modules needed by other platforms would be imported into the modules file, and then selectively executed.

Redundant dependent files will take up the already limited memory in the enclave. When the library OS is checking the integrity of files, it often uses a Merkel tree [6], [15] to cache check information and improve reverification. The more files there are, the more verification information is needed to be cached so that less physical memory can be used by the application. This will cause costly page faults that significantly degrade performance.

Dependencies of Python applications can be constructed accurately by capturing system calls. Linux provides an elegant mechanism for intercepting system calls: `ptrace`, which is named for process tracing. It provides the ability for a parent process to observe and control the execution of child processes so that the monitor process can extract dependency from parameters and return values of system calls in child processes.

[2]http://furius.ca/snakefood

[3]https://www.pyinstaller.org

III. RELATED WORK

As important research progress in the field of trusted computing, SGX plays an important role in system security, especially in cloud computing.

Besides library OS development [4]–[6], SGX has been used in a number of specific applications. VC3 [16] runs MapReduce jobs in SGX enclaves. SGX-FS [17] leverages SGX data sealing capabilities for securing in-memory and persistent storage. It combines the FUSE (Filesystem in Userspace) framework with SGX to protect user data. As indicated by OBLIVIATE [15], all existing SGX filesystems are vulnerable to either system call snooping, page fault, or cache-based side-channel attacks. To address these security limitations, OBLIVIATE redesigns the conceptual components of ORAM (Oblivious RAM) for SGX environments. Similar to a library OS, it seamlessly supports an SGX program without requiring any changes in the application layer. All of the applications were developed with C/C++.

Another solution to trusted computing is the TrustZone [18] proposed by the ARM. TrustZone multiplexes a physical processor into two logical processors in time-sharing to protect data security by isolating them. TrustZone's security subsystem contains all modules from the underlying hardware to the upper operating system. TrustZone has an independent operating system and development environment. The large TCB of TrustZone makes it vulnerable to attackers. Compared with TrustZone, the security boundary of SGX is controllable, and the development environment is friendlier. Even if an attacker gains the privileged operating systems, he cannot steal or tamper with the contents of an enclave.

The program analysis is widely used in the analysis and defense of operating system security, such as auditing by leveraging the rich VM context provided by virtual machine introspection (VMI) [19], building a `ptrace`-based framework to enable rapid prototyping of file systems [7], and securing applications written in C with running it in SGX through a source-level partitioning framework [20].

The dependency analysis method in this paper is inspired by the Glamdring [20] framework. In this framework, Glamdring uses the static analysis function provided by the LLVM compiler to separate the code, but users must first annotate sensitive operation procedures before using the framework to partition the code.

IV. SGXPYTHON

In order to protect the runtime integrity of unmodified Python applications in SGX, we first dynamically collect dependent files using SGXPy, then utilize a modified library OS to verify and execute the application according to the manifest. We assume the runtime environment is secure and complete during the collecting phase.

A. Dynamic Construction of Dependency

We utilize `ptrace` interposition to trace system calls invoked by the user program, and then extract dependencies into a manifest. SGXPy executes and traces the Python application in the native operating system. Algorithm 1 shows the process of extracting dependent files:

1) Launch and Wait. The Python application (*tracee*) is first forked. It then calls `PTRACE_TRACEME` to indicate that it permits the SGXPy process (*tracer*) to track itself. In the tracer process, the `wait` system call is invoked to wait for state change of the tracee process. When being notified, SGXPy uses the `PRACE_GETREGS` request to inspect and extract the information of files accessed from the tracee's registers. The tracer then uses the `PTRACE_SYSCALL` to restart the stopped tracee.
2) Extract. File system calls in the Linux operating system include `read`, `write`, `close` and `seek`. They all start with an `open` system request. The prototype of the `open` function is `int open(const char *pathname, int flags)`. Dependency files can be extracted from the `pathname` parameter. The return value of `open` is a file descriptor, which is used in subsequent system calls.
3) Filter. We note that not all `pathnames` correspond to dependency files. Python sequentially searches the corresponding file according to the path specified in the `PATH` environment variable. The `open` call often ends with `ENOENT` error. We consider the call valid only when it returns a nonnegative value.
4) Write permission processing. SGXPy can distinguish read and write operations from the access mode requested in the `flags` parameter when opening a file. If the file has been modified and the `O_CREAT` flag is set, it will be grouped into the `AllowedFiles` entry, whose files are mapped, but not verified. If a file has not been modified, it will be grouped into the `TrustedFiles` entry, and the hash value of these files will be verified in runtime. In order to minimize file access permissions, we only map the file path when a file is read and written at the same time.

After collecting dependency files into the manifest, SGXPy will calculate the common root directories of dependency files and set mount points for the library OS. Finally, SGXPy uses the AESM (SGX platform service) to sign the hash values of all these dependency files with Intel SGX hardware. Thus, the library OS can validate the integrity of Python applications and runtime environment by comparing the hash of dependencies files between the manifest and the runtime.

Our framework does not provide completeness warranties, since new files may be loaded by an unexecuted branch of code. We mitigate this by providing options for users to manually specify dependency files. The user can also configure in the manifest to allow the program to continue running, and the library OS will prompt missed file information.

B. File access control in multiprocessing applications

By default, the operating system automatically closes file descriptors with the `cloexec` flag before creating a child process. When creating a child process through the library OS, we cannot merely share the file descriptors table of the parent process to the child process. To enhance access control, SGXPy uses the manifest to set up a separate list of accessible files for each process, so that the file access behavior across parent-child processes in enclaves is consistent with the default.

The Linux operating system provides `execve` and `clone` system calls to create processes in user-space. The differences among them are passed parameters and shared resources.

We note that file access permissions may change during successful system calls. For example, if the `FD_CLOEXEC` flag in `fcntl` or `O_CLOEXEC` in `open` call is set, the file will automatically be closed after the `exec` functions are called. Thus, file descriptors in the parent process should not be shared to the newly created process in this situation.

In order to address this mismatch, SGXPy constructs a process tree by hooking system calls involving process life cycle. To reflect the relationship among the parent and child processes, the parent process refers manifests of each child process in its manifest. Thus, when a library OS loads and

Algorithm 1: Extract dependency flow

```
Data: Python Scriptfile
Result: A manifest file for integrity check in SGX
syscalls ← Collect system call information using ptrace;
callinfo ← Filter invalid system calls from syscalls;
foreach syscall in callinfo do
    f ← file arg in open call ;
    if syscall = open then
        if O_CREAT ∈ open flag then
            AllowedFiles ← f;
        else
            TrustedFiles ← f ;
        end
    else if syscall = clone then
        ProcInfo[pid].children.append(subproc);
          subproc.pid ∈ clone call ;
    else if syscall = execve then
        ProcInfo[pid] = procname;
          pid, procname ∈ execve call ;
    end
end
TrustedFiles ← TrustedFiles\AllowedFiles;
Function signtoken(pid)
    foreach child in ProcInfo[pid].children do
        signtoken(child.pid)
    end
    Sign associated manifest files;
    Get token from the AESM service;
end
foreach proc in ProcInfo do
    signtoken(proc.pid)
end
```

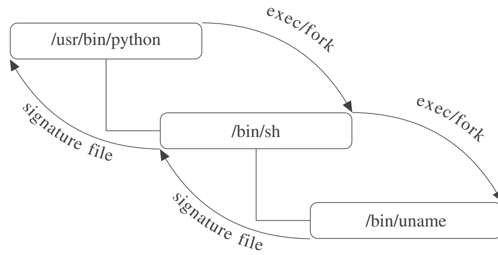

Fig. 1. Sign by the process tree

execute the `fork` operation, it can validate the newly created child process.

Another case of file permission change is the iterative process creation. SGXPy will preserve the parent-child relationship regardless of whether child processes are created by a `clone` or an `execve` call. Even if, in the case of a `clone` call, the file descriptors and offset are shared across processes.

When iteratively creating processes, child processes may call `exec` to establish a grandchild process, ridding the parent process of any control. Thus, the process tree can be used to isolate file descriptors of the parent process from the grandchild. SGXPy will eliminate dependency files from the parent process for the `execved` grandchild.

As shown in Fig. 1, after the process tree is generated, the evidence can be collected in the opposite direction. SGXPy will first generate evidence of the process corresponding to the leaf node (`uname`), and then iterate to the `bash` node. The attestation of the Python main application is generated last, after information of all of the descendant nodes has been collected.

File access permissions in threads are treated similarly to processes. Python calls `clone` to create a thread. File access permissions among the parent and child threads are the same. For multithreading applications such as the web server, thousands of threads may be created to respond to concurrent requests. In order to avoid duplicate checks of dependency files, SGXPy aggregates manifests for threads created by the `clone` system call.

V. Evaluation

A. Implementation Details

We selected the Graphene-SGX [6], [21] as the library OS since developing a library OS from scratch is a nontrivial job. Although the Graphene-SGX supports running Python, all dependent files must be specified manually, so it can only run plain programs. SGXPy adds 1814 lines of Python code, C++ code and other utility scripts to Graphene-SGX. The implementation consists of two main components:

1) A dependency dynamic analysis system developed based on `ptrace` interposition.
2) Patches for Graphene-SGX to improve its compatibility with Python, which consists of an archive file system and memory mapping implementation.

All of our evaluations were performed on an Intel(R) Core i7-7700 CPU @3.60GHz with 16 GB RAM (128 MB for EPC), running Ubuntu 16.04.05 LTS.

During the evaluation phase, SGXPy first constructs a manifest file for each test cases. This manifest file can then be verified by the patched Graphene-SGX library OS to validate the integrity of applications in runtime.

It should be noted that the Graphene-SGX is just one of examples to demonstrate the usage of SGXPy. Depending on applications, developers can choose more TCB-friendly solutions such as SCONE [5] or Intel SGX SDK.

B. Micro-Benchmarks

We first evaluate a few system operations that heavily impact the performance of Python applications when running in SGX. Two primary sources of overhead are the costs of importing modules and allocating memory for arrays.

Module import overhead. Fig. 2 (a) shows the overhead for importing modules. Depending on the number of modules, the latencies of `import` statement of Python in SGX range from 2.75 ms (import 10 modules) to 1.28 seconds (import 2000 modules), whereas on native Linux, the latencies of Python applications are all within 1 ms.

The degeneration of `import` statement in SGX is attributable to the fact that Graphene-SGX has to authenticate the file content before importing a module.

Memory allocation overhead. Fig. 2 (b) shows the overhead for creating a NumPy array with different sizes. It should be noted that Fig. 2 uses a standard logarithmic scale. The latency on the native Linux is constant at 0.1 ms.

When the allocated size of an array is larger than 128MB, the overhead of SGX significantly increases, from 0.06 seconds to 0.9 seconds. This is because SGX has limited memory. When applications request larger memory than 128MB, it will lead to frequent swapping and encryption operations.

The Graphene-SGX library OS currently only supports SGX1, while SGX2 adds new instructions that extend support for dynamic memory management inside of an enclave [22]. These features allow user programs more efficiently to adapt to varying programming workloads.

Subprocess creation overhead. To evaluate multiprocessing Python applications running in SGX, we execute experiments in kinds of process creation functions of Python.

As shown in Fig. 3, the SGX implementation exhibits considerable overheads when compared to the unprotected native applications. The overhead of forking processes in SGX results from two aspects: (`i`) The time spent on library OS startup and dependencies files verification. The library OS does not share check information to forked child processes, so child processes have to repeat the integrity check; (`ii`) The time spent launching a library OS every time a new process is created. Regarding the execution time of single-process Python and multiple attestation in multiprocessing applications, we can conclude that single-process Python applications are more suitable to run in SGX.

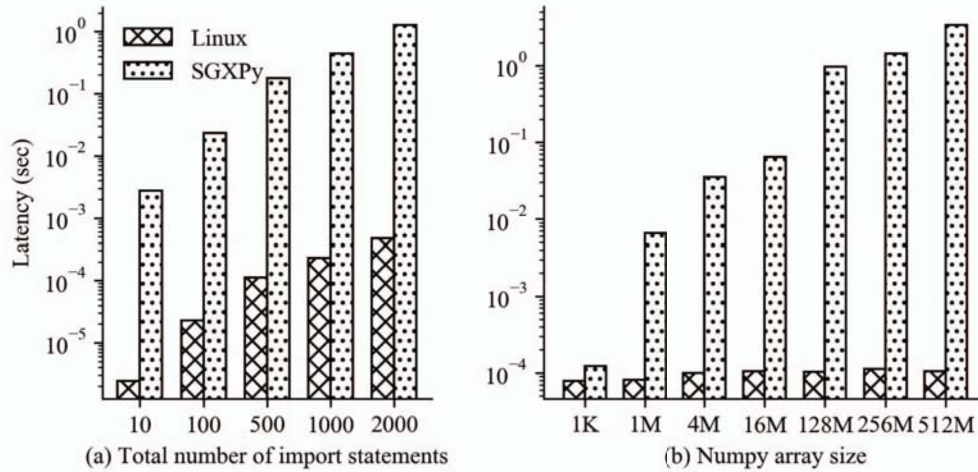

Fig. 2. Micro-benchmarks: Import statement and array memory overhead between native Python and SGX

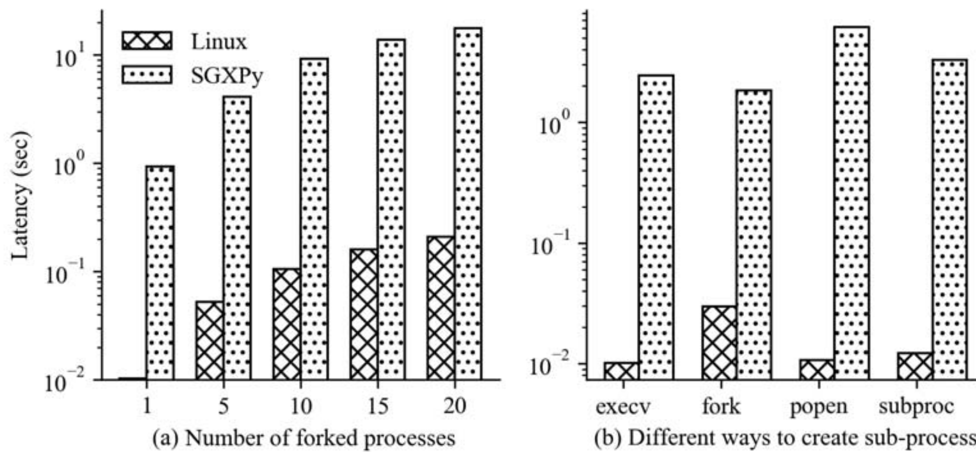

Fig. 3. Micro-benchmarks: Multiprocessing overhead between native Python and SGX

C. Real world applications

This subsection evaluates real-world Python applications run in SGX and the native Linux operating system.

NumPy. We use NumPy's build-in unit tests as the boundary for running sophisticate Python applications in SGX. NumPy includes the `core`, `f2py`, `fft`, `lib`, `linalg`, `ma`, `matrixlib`, `polynomial` and `random` submodules. Other submodules, such as `test`, `doc` and `distutils`, are support modules of NumPy. They are not used by end-user programs and thus ignored in the evaluation. The `f2py` submodule is for running a Fortran application in the Python environment. The submodule dynamically compiles Fortran functions into a module file that Python can import and call directly. It is used less in practice and thus also ignored.

Table II shows the pass rate of the NumPy testing suite in the SGX environment. SGX passed 6299 cases, for a pass rate of 97.60%. This leads us to conclude that SGX has the capability to run almost all of the NumPy functions, even with the isolated environment and limited memory of SGX.

There are four reasons for testing failure displayed by Table III:

1) Encoding failure results from the lack of internationalization support in the library OS.
2) The `ldd` indicates importing rare `.so` files to test compatibility of NumPy with other applications like `Qt` and `Fortran`. Since Numpy is mainly used in scientific computing, the lack of complete support for system features can be tolerated.
3) We note that SGX can only allocate up to 512MB of memory for an array at a time. This limitation causes 2

TABLE II
PASSED TEST COUNT FOR NUMPY SUBMODULES

SubModules	Total tests	SGXPy NumPy passes	Pass rate
core	3403	3402	99.97%
random	138	138	100.00%
ma	440	440	100.00%
lib	1757	1605	91.35%
linalg	187	185	98.93%
fft	28	28	100.00%
polynomial	451	451	100.00%
matrixlib	50	50	100.00%
Total	6454	6299	97.60%

TABLE III
FAILURE CLASS FOR NUMPY TEST CASES

Class	encoding	ldd	large memory	mmap	Total
Count	2	2	2	149	155

large memory test failures. In general, a Python application does not use up an entire array of memory at once. As a result, fragment allocating memory can alleviate this shortcoming of SGX.

4) The 149 (147 for the `test_memmap_roundtrip` test case) failed tests of memory mapping are responsible for the majority of the failures. These failures result from that SGX allocates exclusive memory for each enclave. In order to prevent malware attacks from another enclave, the library OS restricts calling `mmap` for writing a file. We support the opening of a file in read-only mode through the `mmap` call in order to pass other memory mapping test cases, since `mmap` and `read` system calls have the same semantics in this situation.

HTTPServer. We evaluate the latency details of Python `SimpleHTTPServer` using the HTTP benchmarking tool `wrk`[4]. As shown in Fig. 4, we run the benchmark for 30 seconds, using 6 threads, and keep the number of active HTTP connections varying from 100 to 10000.

[4]https://github.com/wg/wrk

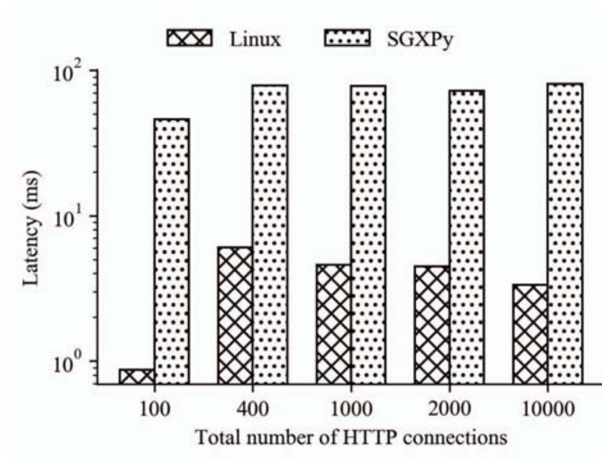

Fig. 4. Latency of web server workloads between native Python and SGX

We can infer that the native operating system is qualified to handle concurrent requests from `wrk`, so the latency occasionally drops as the number of requests increases.

The average latency for processing requests in SGX is about 10× that of native `SimpleHTTPServer`. The latency of the web server in SGX does not increase dramatically with the increase in the number of requests. This is because the measurement of applications also includes the maximum number of threads that can be used in applications. The number of threads has to be assigned in the manifest. The library OS will allocate threads at startup. The web server can use these threads directly when dealing with concurrency requests. Thus, the average latency of handling a single HTTP request in SGX is constant.

Although only a python module `SimpleHTTPServer` is imported in the web server, it requires 135 Python files to run on SGX. It is not practical to specify these dependency files one by one.

In these cases, we can conclude that the benefits of using SGXPy are significant, as it guarantees the integrity of the code, and makes it possible to run sophisticated Python applications unmodified.

It is noteworthy that the design of SGXPy is oriented to an operating system. Although this paper tests only Python applications, other dynamic languages such as R and Ruby can also run on SGX through the framework.

VI. CONCLUSION

In this paper, we present SGXPy, a framework for helping developers make use of SGX for Python applications. SGXPy protects the integrity of unmodified Python applications running in SGX. In the build phase of Python applications, SGXPy utilizes `ptrace` and underlying system calls to automatically and accurately find all dependent files. During the runtime, the Python application is executed through a modified library OS that validates the integrity of the entire runtime environment according to the signed hash in a manifest file.

The evaluation results show that the framework achieves a good balance between accuracy and efficiency for sophisticated Python applications. We have also demonstrated that single-process Python applications are more suitable than multiprocessing applications for running in SGX.

Python applications can currently run in SGX without modification. However, there is no guarantee of offline code confidentiality when storing files on a disk. Malicious users can reverse and steal secrets from the offline file. In future work, we plan to use the Intel Protected File System [15] to enhance the confidentiality of Python applications.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science fundation of China (NSFC) Grant 61532001, Tsinghua Initiative Research Program Grant 20151080475, and gift funds from Huawei, Ant Financial and Nanjing Turing AI Institute.

REFERENCES

[1] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: a structure for efficient numerical computation," Computing in Science & Engineering, vol. 13, no. 2, p. 22, 2011.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, and others, "Scikit-learn: Machine learning in Python," Journal of machine learning research, vol. 12, no. Oct, pp. 2825–2830, 2011.

[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). Broomfield, CO: USENIX Association, 2014, pp. 267–283.

[4] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in ACM SIGARCH Computer Architecture News, vol. 39. ACM, 2011, pp. 291–304.

[5] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association, 2016, pp. 689–703.

[6] C. C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in 2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017, pp. 645–658.

[7] R. P. Spillane, C. P. Wright, G. Sivathanu, and E. Zadok, "Rapid file system development using ptrace," in Proceedings of the 2007 workshop on Experimental computer science - ExpCS '07. San Diego, California: ACM Press, 2007, pp. 22–es.

[8] G. Wang, S. Xiang, Y. Duan, L. Huang, and W. Xu, "Do Not Pull My Data for Resale: Protecting Data Providers Using Data Retrieval Pattern Analysis." in SIGIR, 2018, pp. 1053–1056.

[9] R. Amankwah, P. K. Kudjo, and S. Y. Antwi, "Evaluation of Software Vulnerability Detection Methods and Tools: A Review," International Journal of Computers and Applications, vol. 169, no. 8, pp. 22–27, 2017.

[10] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," arXiv preprint arXiv:1801.01203, 2018.

[11] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, 2017, pp. 3–24.

[12] D. Cai and M. Kim, "An empirical study of long-lived code clones," in International Conference on Fundamental Approaches to Software Engineering. Springer, 2011, pp. 432–446.

[13] R. Silva, P. Barbosa, and A. Brito, "DynSGX: A Privacy Preserving Toolset for Dynamically Loading Functions into Intel (R) SGX Enclaves," in 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE, 2017, pp. 314–321.

[14] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy, vol. 13. ACM New York, NY, USA, 2013.

[15] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX," in Proceedings 2018 Network and Distributed System Security Symposium. San Diego, CA: Internet Society, 2018.

[16] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy Data Analytics in the Cloud Using SGX," in 2015 IEEE Symposium on Security and Privacy. San Jose, CA: IEEE, May 2015, pp. 38–54.

[17] D. Burihabwa, P. Felber, H. Mercier, and V. Schiavoni, "SGX-FS: Hardening a File System in User-Space with Intel SGX," in 2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Dec. 2018, pp. 67–72.

[18] B. Yang, K. Yang, Y. Qin, Z. Zhang, and D. Feng, "DAA-TZ: An Efficient DAA Scheme for Mobile Devices using ARM TrustZone," in IACR Cryptology ePrint Archive, 2015.

[19] F. Jiang, Q. Cai, L. Guan, and J. Lin, "Enforcing Access Controls for the Cryptographic Cloud Service Invocation Based on Virtual Machine Introspection," in International Conference on Information Security. Springer, 2018, pp. 213–230.

[20] J. Lind, C. Priebe, D. Muthukumaran, D. OKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX," in 2017 USENIX Annual Technical Conference (USENIX ATC 17). Santa Clara, CA: USENIX Association, 2017, pp. 285–298.

[21] C. C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-process Applications," in Proceedings of the Ninth European Conference on Computer Systems, ser. EuroSys '14. New York, NY, USA: ACM, 2014, p. 9.

[22] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel software guard extensions (intel sgx) support for dynamic memory management inside an enclave," in Proceedings of the Hardware and Architectural Support for Security and Privacy 2016. ACM, 2016, p. 10.