# mShield: Protecting In-process Sensitive Data Against Vulnerable Third-Party Libraries

Yunming Zhang[1], Quanwei Cai[2], Houqiang Li[1], Jingqiang Lin[1(✉)], and Wei Wang[3]

[1] University of Science and Technology of China, Hefei 230027, Anhui, China
zhangyunming@mail.ustc.edu.cn, {lihq,linjq}@ustc.edu.cn
[2] Beijing Zitiao Network Technology Co., Ltd., Beijing 100190, China
[3] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100085, China
wangwei@iie.ac.cn

**Abstract.** Third-party libraries (TPLs) are widely adopted in softwares for integrating special functions (e.g., compression) efficiently. However, as most TPLs are in the same process with the invoker, attackers could exploit memory disclosure vulnerabilities in TPLs to read the sensitive memory data of the victim process. Therefore, once a vulnerability found in a TPL, all softwares with this TPL need to be patched in time, which is impractical. In this paper, we propose a cryptography-based isolation (named *mShield*) between the data memory of the invoker and TPLs, to prevent TPL vulnerabilities from being exploited to read the invoker's sensitive memory data. mShield performs a user-mode and lightweight memory analysis, figures out the invoker's memory space (including stack, heap, user-defined ones in BSS/data segment), encrypts them before invoking any TPL function, and automatically decrypts them once the function returns, without interrupting the normal execution. mShield performs the encryption/decryption in the trusted environment provided by Intel SGX, which prevents the attacker from reading the cryptographic key, and alerts (i.e., the invoker's decryption fails) in time once the encryption context is tampered with (e.g., by illegal invocations of decryption). We have implemented mShield, and adopted it to protect Nginx against a potentially vulnerable TPL (i.e., zlib). The experiment demonstrates mShield's effectiveness (TPLs fail to read the invoker's plaintext sensitive memory data) and acceptable efficiency (about less than 4× time cost).

**Keywords:** Memory Disclosure · Memory encryption · Intel SGX · Isolation

The work was partially supported by National Key RD Plan of China under award No. 2020YFB1005803.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2025
Published by Springer Nature Switzerland AG 2025. All Rights Reserved
H. Duan et al. (Eds.): SecureComm 2023, LNICST 567, pp. 496–513, 2025.
https://doi.org/10.1007/978-3-031-64948-6_25

## 1 Introduction

Third-party libraries (TPLs) are widely employed to facilitate software development, as shown in [34], more than 80% softwares have TPLs integrated. However, vulnerable and outdated TPLs, one of OWASP top 10 security risks, introduce significant information leakage risks, as TPLs, either statically or dynamically linked, are in the same process with the invoker, which allows the attackers to exploit TPLs' memory disclosure vulnerabilities to read process's *whole* memory (including the invoker's sensitive data) [9,19,27].

The in-process memory disclosure attacks based on vulnerable TPLs, are easy to initiate and difficult to prevent. Attackers could exploit one TPL's vulnerability to attack all softwares which have integrated this TPL, even not need to modify the invoker's binary and control flows. The in-process attacks make the inter-process isolations [4,12,23,31] and inter-process memory encryption [5,18] ineffective. Moreover, as running in the same address space with the victim, vulnerable TPLs could be exploited to bypass existing binary and control flow integrity protections, such as data execution prevention (DEP) [29], address space layout randomization (ASLR) [15], control flow integrity (CFI) [1,35], and code pointer integrity (CPI) [24]. For example, the OpenSSL Heartbleed vulnerability [14] is exploited to copy in-process memory data (e.g., the invoker's cryptographic keys) and send it as regular TLS heartbeat responses to remote adversaries.

Various mitigations against in-process memory disclosure attacks have been proposed. These solutions construct the in-process memory isolation to prevent illegal read on the sensitive memory, and can be classified into hardware-based [9,16,26] and software-based [3,6,25] ones. The hardware-based solutions utilize the hardware features to ensure the memory unit is only accessed by a specified processing unit. For example, IMIX [16] introduces a self-defined hardware feature to build an isolated memory page that can only be accessed by a simulated instruction `smov`, Shreds [9] relies on ARM memory domain mechanism [2] to ensure the protected memory pool is only accessed by a developer-defined segment of a thread execution, SeCage [26] adopts the hardware virtualization to achieve the isolation within an application. The software-based solutions construct the isolated memory domain based on OS primitives or the extensions of runtime systems. For example, Wedge [3] introduces new OS primitives to implement a variant of Linux processes for the least-privilege partitioning (i.e., each thread can only access the necessary memory compartments), Light-weight contexts (lwCs) [25] proposes a new OS abstraction (named lwCs, orthogonal to threads) to achieve in-process memory isolation by maintaining the independent units of protection, privilege, and execution state within a process, and DataShield [6] provides compiler annotations for the programmers to specify sensitive variables and modifies runtime systems (i.e., libc++) to create two separate memory regions for the sensitive and non-sensitive memory objects.

However, obstacles exist for these solutions to be practically deployed on commercial-off-the-shelf (COTS) platforms, as they fail to satisfy the following requirements simultaneously:

1. Compatible with COTS platforms. The solution will be easily deployed if it (1) does not rely on any hardware feature or only the widely deployed ones, and (2) requires no customizations on COTS platforms. However, IMIX [16] requires a self-designed and not-deployed hardware extension, Shreds [9], Wedge [3], lwC [25], and DataShield [6] need to modify OS and/or runtime systems, while SeCage [26] needs to customize the hypervisor, which makes them fail to be deployed when the hardware, OS and hypervisors are out-of-control.
2. Developer-friendly. The solution is expected to be easily integrated for the developers. For example, the developer may only need to invoke a function before and after invoking the invoked TPL functions, instead of manually figuring out all the memory unit related with the sensitive variables [6,16] nor splitting the applications [3,9,26].

In this paper, we propose a cryptography-based solution (named *mShield*) to shield the invoker's (sensitive) memory from vulnerable TPLs. To achieve this, mShield provides (1) a user-mode and lightweight memory analysis, to figure out the invoker's memory (including stack, heap, user-defined ones in BSS/data segment), and (2) a library for the developer, which encrypts all the invoker's memory (except the ones required by the TPL) before invoking the TPL function and decrypts them once the function returns. Therefore, mShield is easy to be integrate with, as the developers only need to invoke the encryption and decryption APIs before and after invoking the TPL, instead of manually identifying all the sensitive memory (including the intermediate variables) or re-compiling/re-structuring the TPLs. As the memory analysis doesn't rely on any assumptions on memory disclosure attacks and TPLs, mShield is effective against unknown memory disclosure vulnerabilities in TPLs.

As focusing on mitigating against in-process memory disclosure attacks on COTS platforms, mShield does not modify OSes, hypervisors, runtime systems nor TPLs, and performs the memory analysis, encryption and decryption in the user mode. To obtain the correct memory layout and execute encryption/decryption exactly, mShield assumes the OS is trustworthy, which is reasonable, especially when the application is deployed on mainstream cloud services. Another challenge for the user-mode solution is that the attackers might attempt to read the cryptographic key or obtain the sensitive memory variables by invoking the decryption APIs. The mShield solution performs the encryption, decryption and cryptographic contexts management in the widely deployed trusted environment (e.g., Intel SGX [11]), which prevents the attackers from obtaining the cryptographic key and halt the process execution once the decryption is invoked illegally.

We summarize our contributions as follows.

– We provide a solution to mitigate against in-process memory disclosure attacks due to vulnerable TPLs, which can be directly deployed in COTS platforms, and is easy to be integrated with for developers.

– We have implemented the prototype, and integrated it with Nginx (as the invoker) to protect from the potentially vulnerable zlib (as the TPL), for the evaluations of effectiveness and performance.

The rest of this paper is organized as follows. We introduce the background and related work in Sect. 2, provide the threat model and system design in Sect. 3, and illustrate the implementation details in Sect. 4. We then present the security analysis and performance evaluation in Sect. 5, discuss the limitations in Sect. 6, and conclude the paper in Sect. 7.

## 2 Background and Related Work

### 2.1 Intel SGX

Intel SGX is a widely deployed hardware feature, and provides user-level isolated execution environments (i.e., enclave). The enclave memory are hardware encrypted to store in a reserved memory region (i.e., enclave page cache) and decrypted on the fly within the CPU. Therefore, SGX ensures no other code (including OS and hypervisors) except the code residing within enclave memory can access the plaintext enclave data, which protects the confidentiality of enclave data. An SGX application is divided into two logical components: trusted one and untrusted one. The trusted one is the enclave and suggested to be as small as possible (e.g., limited to operations on secret data), while untrusted one is the rest of the applications. SGX provides enclave call (eCall) to enter the enclave, and outside call (oCall) to call an untrusted function (e.g., system calls and I/O operations) from an enclave [28].

### 2.2 Related Work

**Fine-Grained Access Control.** The solutions [3,6,9,16,25,26,33,36] grant only special parts of the application the access permissions of the sensitive variables, to mitigate the in-process memory disclosure attacks. IMIX [16] only allows a simulated instruction `smov` to access the sensitive memory page. Wedge [3] grants the access of the data to the code in the same thread. ARMlock [36] recompiles the untrusted module (e.g., TPL) and limits the memory it could access based on ARM memory domain mechanism, NativeGuard [33] provides similar functionality in Android. SeCage [26], Shred [9] and lwCs [25] split the target application into multiple compartments where the data can only be accessed by the code in the same compartment, based on hardware virtualization, ARM memory domain mechanism and a new OS abstraction, respectively. DataShield [6] extends C/C++ compiler to allow the programmers to specify the access control policies of sensitive variables, which is enforced by the modified runtime systems (e.g., libc++). However, these works either need to recompile the TPL [36], modify OS and/or runtime systems [3,6,9,25], customize the hypervisor [26], or rely on special hardware supports [16], and therefore cannot

be directly deployed on COTS platforms. mShield ensures only the invoker could access the sensitive memory by encrypting invoker's whole memory before invoking TPLs' functions, and needs no customizations on COTS platforms, which allows it to be deployed directly.

**Memory Encryption.** mShield adopts memory encryption to prevent in-process memory disclosure attacks. Various memory encryption solutions [5,7,10,13,17–19,21,22,30] have been proposed to prevent the memory disclosure attacks [8,20,32]. However, these works fail to mitigate in-process memory disclosure attacks as the decryption is performed automatically when the in-process code read the sensitive variables. Moreover, these solutions need to modify the OS [13,18,21,30], or hypervisor [7,17,22], or rely on special hardware feature [19]. mShield needs no modifications on OS, runtime systems nor hypervisor, and relies on the widely deployed SGX to perform encryption/decryption and manage the cryptographic context, which alerts in time when the adversaries attempt to read the plaintext sensitive data by maliciously invoking the decryption call.

# 3 System Design

We present the threat model and design overview of mShield in this section.

## 3.1 Threat Model and Assumptions

In mShield, we assume the attackers have successfully performed the in-process memory disclosure attacks via the vulnerable or malicious TPLs. The attackers' goal is to access the sensitive data in the victim program's virtual memory space. To achieve this, the attackers could (1) exploit TPLs' vulnerabilities to read the victim invoker's encrypted memory, or even (2) entice the victim programs' developers to integrate the malicious TPLs, which invoke the TPL's decryption function and then read the invoker's plaintext sensitive memory. The victim programs may integrate vulnerable TPLs (e.g., the unpatched one, or the one from untrusted entities), and TPLs' vulnerabilities are expected to be unknown to the developers of victim programs, considering the existence of zero-day vulnerabilities and the ones found after the release of the victim program.

We assume the OS and Intel SGX are trustworthy, which serve as the TCB for mShield. The same as Shred [9], the assumption of trustworthy OS is reasonable, as the in-process memory disclosure attacks which mShield aims to prevent, will become unnecessary when the attackers have already control the OS. The assumption that SGX enclave memory cannot be accessed outside the enclave, is widely adopted in SGX applications, and is considered practical when the countermeasures to known SGX vulnerabilities have been deployed.

mShield focuses on protecting victim invokers from the data leakage caused by vulnerable TPLs. And, we assume that the invoker program (1) has been well protected from control-flow hijacking attacks, preventing the attackers from reading the sensitive data by invoking the invoker's function, and (2) processes the sensitive data itself and never passes it to TPLs (that is, cryptographic

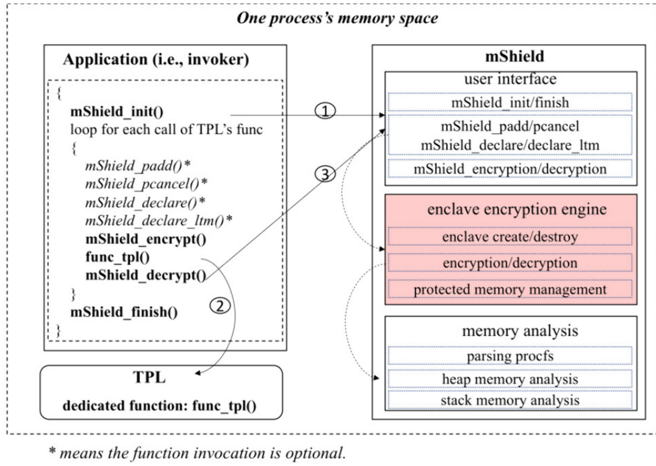


**Fig. 1.** Overview of mShield.

libraries, such as OpenSSL, are not considered in this paper). Currently, mShield does not maintain the contexts to distinguish multiple simultaneous invocations of TPLs' functions, and therefore limits the invoker program to invoke TPLs' functions synchronously. This limitation could be eliminated and will be further discussed in Sect. 6.

### 3.2 Design Overview

The goals of mShield include (1) protecting the application (i.e., invoker) from the in-process memory disclosure attacks caused by vulnerable TPLs, (2) requiring no customizations on COTS platforms nor special (not widely deployed) hardware features to achieve direct deployment on existing COTS platforms, and (3) developer-friendly interfaces for easy integration.

To achieve these goals, mShield encrypts the invoker's memory using the widely deployed hardware feature (Intel SGX) before invoking TPL functions to prevent adversaries from reading the invoker's plaintext memory via TPL's vulnerabilities (Goal 1), provides a user-mode memory analysis module to automatically figure out the memory needed to protect and completes the encryption/decryption in the trusted environment provided by the widely deployed Intel SGX, without any new hardware feature, nor modification on OS, hypervisor, TPLs and runtime systems (Goal 2), and encapsulates mShield as a library for the developers to integrate through at most 8 function calls (Goal 3).

As shown in Fig. 1, there are three components in mShield, i.e., user interface, SGX enclave of encryption engine, and memory analysis module. The user interface invokes the SGX enclave of encryption engine to complete the enclave creation/destroy, user-defined protected memory management, and invoker's memory encryption/decryption. The SGX enclave of encryption engine performs the

above operations in the enclave, and invokes the memory analysis module to obtain the invoker's in-use memory automatically.

To protect the application from vulnerable TPLs, developers firstly invoke mShield's initialization function (only once for this application), then call the mShield's memory management and encryption functions (Step 1 in Fig. 1) for each invocation of TPL function (Step 2 in Fig. 1), and invoke the mShield's memory management and decryption functions (Step 3 in Fig. 1) on the return of each TPL function, and finally call the mShield's finalization function to destroy the enclave context (only once for this application).

As the application, TPLs and mShield run in the same memory address of the process, the attackers of TPLs might attempt to bypass mShield and perform in-process memory disclosure attacks as follows.

– Read the cryptographic keys of the memory encryption from the mShield's memory address. To prevent this, mShield performs the encryption and decryption in an SGX enclave, which ensures the adversaries cannot obtain the invoker's plaintext memory or the cryptographic keys.
– Invoke mShield's decryption function to obtain the invoker's plaintext memory. To prevent this, the SGX enclave of encryption engine generates a fresh cryptographic key for each encryption and combines the key with this invocation. Then the application will halt as the protected memory obtained by the invoker will be incorrect (either be decrypted twice or encrypted using a different key).

To minimize the integration overhead of mShield, we choose the stream cipher (e.g., AES-CTR) to achieve the in-place encryption. That is, the length of plaintext and ciphertext equals. Therefore, the ciphertext of the memory could be stored directly in the original memory regions, requiring no extra memory for integrating mShield.

The invoker's in-use memory varies on each invocation of TPL functions, therefore mShield analyzes the process's memory layout on the fly, to figure out the invoker's memory to be protected. As mShield assumes the OS is trustworthy and focuses on preventing the in-process memory disclosure attacks, we only need to analyze the user-mode memory components of the invoker process, which includes text segment, mapping segment, stack, heap, BSS segment, and data segment.

We choose to encrypt the invoker's stack and heap memory from the automatic analysis results, the memory regions in BSS and data segment specified by the developers based on the following analysis.

– Text Segment. The text segment contains the code and constant variables, and could be obtained through analyzing the binary. Therefore, mShield excludes the text segment memory for encryption.
– Mapping segment. The mapping segment is shared among multiple processes, and therefore also excluded from the encryption scope.
– Stack. The stack contains sensitive variables of the process, and is chosen to be encrypted. There are four types of stacks in Linux, that is, the process

stack, thread stack, kernel stack, and interrupt stack. The kernel stack and interrupt stack stores the variables when the process works in the kernel mode, and therefore mShield chooses to not encrypt these two memory regions. mShield encrypts the whole process stack except the ones need to be used in the invoked TPL function and that specified as non-sensitive memory by the developers. The thread stack also needs to be encrypted and will be processed in the future work.
- Heap. mShield encrypts all the in-use heap memory of the process, except the ones specified as non-sensitive by the developers. The in-use heap memory is figured out automatically in mShield.
- BSS and Data Segment. BSS segment and data segment contain the global and static variables, and should also be encrypted. However, there is no fixed data structure for these two segments, mShield cannot determine whether the variables will be used in the TPL or not, automatically. Therefore, mShield does not encrypt the BSS segment and data segment in default, and provides the interface for the developers to define the memory to encrypt.

## 4 Implementation

In this section, we first provide the implementation details of mShield which include (1) SGX enclave of encryption engine for managing/encrypting/decrypting the protected memory in the enclave, (2) the memory analysis module for automatically recognizing the memory needed to protect, and (3) the user interface which encapsulates mShield's details and provides the developer-friendly interfaces for easy integration. Finally, we show how to apply mShield to protect applications from vulnerable TPLs.

### 4.1 SGX Enclave of Encryption Engine

mShield provides encryption engine to maintain the invoker's memory to protect and encrypt/decrypt these memory before/after invoking any TPL function. The encryption engine is an enclave, which ensures the confidentiality of the cryptographic keys, prevents adversaries from tampering with the memory scope to encrypt, and detects illegal invocations of decryption.

The encryption engine provides the following types of functionalities.

- Encryption and decryption. mShield chooses AES-CTR to perform the in-place encryption, that is, the ciphertext is directly stored in the memory regions of the plaintext as the length of the ciphertext and plaintext equals. mShield completes the encryption and decryption only using Intel SGX SDK, without any other dependency. In details, mShield uses `sgx_read_rand()` to generate different 128-bit IVs and keys for encrypting protected heap, stack and data in the BSS and data segment, completes encryption and decryption by invoking `sgx_aes_ctr_encrypt()` and `sgx_aes_ctr_decrypt()`. On each invocation of a TPL function, mShield uses one eCall to invoke the enclave

encryption. The enclave encryption uses one oCall to invoke the memory analysis for the memory needed to be protected automatically, then combines the result with the user-defined one, generates and stores a fresh IV and key, and completes the encryption. On the return of a TPL function, mShield uses one eCall to invoke the enclave decryption, which returns the decrypted memory and destroys the cryptographic context (e.g., IV and key).
– Protected memory management. mShield maintains the protected memory defined by the application developers in the enclave, and provides 4 eCall to specify these memory regions, two (i.e., add and cancel) for defining the memory data to be encrypted, two for declaring the permanent and one-time memory not to encrypt.

### 4.2 Memory Analysis

mShield provides the user-mode memory analysis, to figure out the invoker's memory regions that need to be encrypted before invoking the TPL functions. The memory analysis first parses the process filesystem to obtain the starting and ending address of heap and stack segments, and then refines the in-use heap and stack.

– Parsing `procfs`. mShield traverses the maps file in the process filesystem (i.e., `procfs`) to find the starting and ending address of the heap and stack segment. As this process needs to read the virtual files in `procfs`, it is not included in the enclave and invoked by the encryption engine (in the enclave) through oCall. `Procfs` is a virtual file system maintained by the kernel to store each process's information, and each line in the maps file includes the pathname of each memory mapped file and the virtual address of the memory. The virtual address specifies the starting and ending addresses, which are separated using a '-' symbol. We distinguish the heap and stack memory region through the pathname, which includes the word 'heap' and 'stack' to indicate the type of memory.
– Figuring out in-use heap chunks. mShield finds all in-use heap memory by traversing all the heap memory based on the `malloc_chunk` structure (as shown in Fig. 2) which is used by the kernel for heap management. The heap is a continuous memory space, and split into consecutive chunks by the memory allocator (e.g., glibc) using the functions `brk()` and `sbrk()`. We start from the start address of the heap, iterate all the chunks based on chunk size, and classify the chunks into in-use and free ones based on the flag bits in `mchunk_size` field of the `malloc_chunk` structure, where the flag bit 'P' indicates the chunk is in-use.
– Figuring out stack memory to encrypt. After obtaining the starting and ending address of the stack, mShield determines the stack's bottom address (fixed at the creation of the stack) easily, gets the stack's the current top pointer from the RSP register at the execution of the memory analysis function (invoked by the encryption function), then excludes the memory (40 bytes) that cannot be encrypted, and finally figures out the stack memory to be

```
struct malloc_chunk {
    INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
    INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */
    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;
    /* Only used for large blocks: pointer to next larger size.  */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
    struct malloc_chunk* bk_nextsize;
};
```

**Fig. 2.** The structure of heap chunk in Ubuntu 16.04.

encrypted. In mShield, the 40-bytes stack memory that needs to be excluded from encryption, includes 8-bytes return address of the encryption function, 4-bytes local variable, 8-byte canary value, 8-bytes RSP register value and 4 bytes for stack alignment. Moreover, the variables used by TPLs need to be excluded from encryption (otherwise the TPL functions cannot be executed). mShield provides `mShield_declare` and `mShield_declare_ltm` for the developers to declare these variables.

### 4.3 User Interfaces

For easy integration, mShield encapsulates the functions in the SGX enclave, and provides eight interfaces.

- `mShield_init` and `mShield_finish`. These two functions need to be invoked at the beginning and ending of the application, to setup and cleanup the SGX enclave context. `mShield_init` calls `sgx_create_enclave` to load the enclave image file, obtains and records the enclave Id on a successful load, uses the enclave Id to invoke the heap memory analysis function through eCall to obtain the already-used heap chunks which are not sensitive data of this application and will be stored in the blacklist to exclude in the memory encryption. The heap memory analysis function reads and analyses the maps file of this process in the proc filesystem through oCall, and traverses the entire heap memory to find all the in-use chunks. `mShield_finish` invokes `sgx_destroy_enclave` to unload the enclave (specified by the enclave Id).
- `mShield_encrypt` and `mShield_decrypt`. These functions are invoked before and after invoking any TPL function sequentially. `mShield_encrypt` obtains the current stack top pointer and performs the in-place memory encryption, while `mShield_decrypt` completes the in-place memory decryption, where the memory encryption and decryption operations are executed in the enclave through eCalls.
- `mShield_padd` and `mShield_pcancel`. mShiled maintains the user-defined whitelist of memory regions in the enclave. The memory in the whitelist will be encrypted in addition to the automatic memory analysis results. The

developers call `mShield_padd` to add the target regions of BSS and data segment to the whitelist, and delete the regions from the whitelist through `mShield_pcancel`.
- `mShield_declare` and `mShield_declare_ltm`. The mShiled maintains the user-defined permanent and temporary backlists of memory regions in the enclave. The memory in the blacklists will be excluded from the automatic memory analysis results and never be encrypted. The developers identify the long-term non sensitive memory regions in heap and stack, and add them to the permanent backlist through `mShield_declare`, which will never be encrypted during the whole life cycle of this process. `mShield_declare_ltm` allows the developers to specify one-time non sensitive memory regions in heap and stack, which will be stored in the temporary backlist and cleared once `mShield_decrypt` is called.

`mShield_init`, `mShield_encrypt`, `mShield_decrypt` and `mShield_finish` always need to be invoked. While `mShield_padd`, `mShield_pcancel`, `mShield_declare`, and `mShield_declare_ltm` are invoked optionally by professional developers to specify the protected memory regions, for example, to improve the performance.

### 4.4 Application

To demonstrate mShield is easy to integrate, we provide integration details for two typical types of applications: (1) one is the simple application who passes the parameters to the TPL functions and obtains the processing results, and (2) the other is the complicated one (e.g., Nginx) who manages the memory by itself for better performance and allows the TPL functions to call the invoker's function (i.e., a callback function as the parameter) for asynchronous processing.

For simple applications, the developers only need to invoke `mShield_init`, `mShield_encrypt`, TPL functions, `mShield_decrypt` and `mShield_finish` sequentially. To simulate the simple application, we construct a demo which uses zlib (as TPL) for file compression. After integrating mShield as above, the demo works correctly, and the effectiveness of mShield is also checked as the invoker's memory remains encrypted during the execution of zlib's functions.

mShield could also be easily adopted to protect complicated applications against vulnerable TPLs. We choose Nginx, a widely deployed open-source web server, as the invoker, and zlib, a widely used compression library, as the TPL. The integration only needs to add less than 80 lines in Nginx. Same as most industry applications, Nginx has two features which need to be processed during the integration.

- Memory pool. Nginx manages the memory by itself, to avoid the frequent invocations of heavy `malloc` functions. Then, in mShield, the unused heap memory may be treated as the used one and encrypted/decrypted before/after invoking the TPL functions, as mShield distinguishes the used and unused heap memory based on whether it has been assigned. This mistake affects the

normal execution, for example, the function `deflate` in zlib directly writes the *plaintext* compression result in the heap to return it to Nginx. The unnecessary decryption on the plaintext compression result, makes Nginx obtain the incorrect result. To solve this problem, the developer only needs to invoke `mShield_declare_ltm` to exclude the heap memory regions which are unused, which avoids unnecessary encryption/decryption.

– Callback function. Nginx provides the callback functions (i.e., memory allocation and release) to `deflateInit2` and `deflateEnd` in zlib, to allocate/release memory from Nginx's memory pool. Then, in mShield, these callback functions are invoked by zlib and executed before Nginx calling the `mShield_decrypt` function. This execution fails as the variables of callback functions are still encrypted. To solve this problem, the developer only needs to invoke `mShield_declare_ltm` to exclude the memory that required by callback functions, which ensures the memory remains plaintext during the execution of callback functions.

## 5 Analysis and Evaluation

### 5.1 Security Analysis

We demonstrate that mShield prevents the in-process memory disclosure attacks caused by vulnerable TPLs from two aspects, that is, whether the adversaries tamper with mShield. Finally, we provide an experiment to show the effectiveness of mShield.

Firstly, the attackers fail to obtain the invoker's plaintext sensitive data when they do not attempt to tamper with mShield. mShield is a library which also the developers to encrypt the invoker's heap, stack and data segment in process space, before each invocation of TPL functions, which ensures that TPL's code (assumed to be controlled by the adversary) can only read the ciphertext of the invoker's memory. Also, the adversaries cannot infer the modification of sensitive data from multiple ciphertexts. The modification of sensitive data may also be sensitive, and adversaries might attempt to infer it by analysis the difference of the continuous ciphertexts. This attack always fails in mShield, as we re-generate the cryptographic key for each memory encryption, which makes the difference among the continuous ciphertexts be random.

Secondly, the attackers cannot infer the invoker's sensitive memory, even when they attempt to make mShield behave unexpectedly. The adversaries might attempt to (1) make the invoker's sensitive memory not encrypted, (2) read mShield's cryptographic key to decrypt the ciphertext of memory, or (3) invoke mShield's decryption function for invoker's plaintext memory without being found. However, these attempts always fail in mShield as follows.

– The adversaries fail to control the memory regions to be encrypted, as (1) the trustworthy OS provides the correct memory space information of the targeted process, (2) mShield is trustworthy and verified comprehensively to perform the analysis correctly, and (3) finally the analysis results are stored in Intel SGX to prevent the malicious modification.

- The adversaries cannot obtain mShield's cryptographic keys for the memory encryption. To be compatible with COTS platforms, mShield works in the same virtual memory space with the invoker and TPL. The adversaries might attempt to read the cryptographic key for decrypting the ciphertext of sensitive data. This attack cannot succeed, as the cryptographic key is generated and used in an Intel SGX enclave, which ensures the key is only accessed by the code in the enclave.
- The adversaries fail to invoke mShield's decryption function without being detected. The TPL code might attempt to invoke the mShield's decryption function to obtain the plaintext sensitive data directly, and then re-encrypt the memory through one encryption function call. However, mShield generates a fresh cryptographic key for each memory encryption, and destroys the key after the completion of the decryption. Therefore, this malicious invocation will be easily found, as the decryption called by the invoker will fail (when the attacker only invokes the decryption) or return incorrect values (when the attacker invokes both the decryption and encryption). In this case, the invoker will halt automatically.

**Experiment Verification.** We have performed the experiments to dump the process's memory before invoking the TPL functions and at the point of executing the TPL functions, with different sizes of the invoker's heap, stack and global memory. The experiments demonstrate that mShield correctly encrypts the invoker's memory before invoking the TPL functions and decrypted only after TPL functions returns. Figure 3 provides a screenshot for the heap memory before and during the execution of the TPL function, which is set as the plaintext 'Q' and encrypted to be random characters during the execution of the TPL function.

### 5.2 Performance Evaluation

We have performed experiment to evaluate the performance overhead introduced by mShield. We use mShield to protect Nginx from potentially vulnerable TPL zlib, the integration details are provided in Sect. 4.4.

**Experiment Settings.** We use two machines in the experiment, one deployed Nginx integrated with zlib and mShield to work as the server, while the other one is used as the client with ApacheBench. The server, with Intel SGX, runs Ubuntu 16.04, with Intel core i7-6700hq CPU (2.60 GHz) and 8 GB RAM. The client runs Windows 10 with Intel Core i7-6700HQ CPU (2.60 GHz) and 8 GB RAM. The two machines are connected with 1000 Mbps LAN.

In the experiment, the client uses ApacheBench to repeatedly request a file from the server with different concurrency. The size of file is 196,903 bytes before compression and reduced to 61,443 after compression. We have performed the experiment 10000 times and calculated average response time, the results are shown in Fig. 4.

To evaluate mShield's performance overhead thoroughly, we have constructed five scenarios as follows.

```
00038b80: 0000 0000 0000 0000 8100 0000 0000 0000  ................
00038b90: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038ba0: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038bb0: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038bc0: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038bd0: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038be0: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038bf0: 5151 5151 5151 5151 5151 5151 5151 5151  QQQQQQQQQQQQQQQQ
00038c00: 5151 5151 5151 5151 3100 0000 0000 0000  QQQQQQQQ1.......
00038c10: 1022 6300 0000 0000 989b 9cf7 ff7f 0000  ."c.............
```

Before encrypt

```
00038b80: 0000 0000 0000 0000 8100 0000 0000 0000  ................
00038b90: 1c4a a689 c75c 1b35 080c fa5d b6b1 7fe4  .J...\.5...]....
00038ba0: 7625 10bf f619 47b1 6227 46cd aea7 12b8  v%....G.b'F.....
00038bb0: b857 9847 1a9f e2b9 3d3a b025 42f1 0918  .W.G....=:.%B...
00038bc0: 05e8 fc57 37cd 73a3 7203 226a 3c73 6f2d  ...W7.s.r."j<so-
00038bd0: 837b 54bb eb6f c255 f3e7 ff66 c4ff 05ca  .{T..o.U...f....
00038be0: 74e3 a2b3 12db aa30 2a07 7426 8117 6006  t......0*.t&..`.
00038bf0: e264 a126 403f 6bdf 4dca 7733 67e2 bacf  .d.&@?k.M.w3g...
00038c00: b775 1545 c983 ecd2 3100 0000 0000 0000  .u.E....1.......
00038c10: 1022 6300 0000 0000 989b 9cf7 ff7f 0000  ."c.............
```

After encrypt

**Fig. 3.** Heap memory before and during execution of the invoked TPL function.

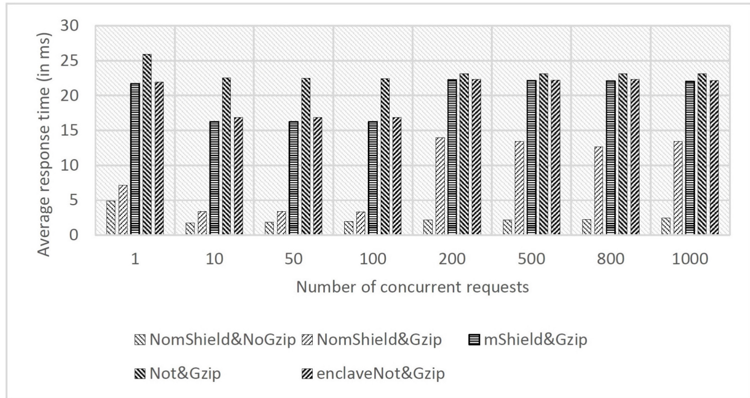


**Fig. 4.** Average response time (in ms).

– NomShield&NoGzip. Nginx is deployed directly, without using the protection of mShield and Gzip of zlib.
– NomShield&Gzip. Nginx is deployed with Gzip of zlib for compression, but not protected using mShield.
– mShield&Gzip. Nginx adopts Gzip of zlib for compression, and uses mShield to protect the invoker's memory from potentially vulnerable gzip.
– enclaveNot&Gzip. Same as mShield&Gzip, except that mShield just sets the inversion of the plaintext as the ciphertext, instead of AES-CTR encryption.

- Not&Gzip. Same as enclaveNot&Gzip, except that the encryption is performed out of enclave.

As demonstrated in Fig. 4, we find that the overhead introduced by mShield is modest (less than 14.54 ms, and less than about 4×), and AES-CTR performs better than the simple inversion, due to the hardware acceleration provided by Intel SGX. Gzip compression saves the data transmitted, but the increases the response time. The main overhead is introduced by the memory analysis, which could be easily reduced when the protected memory is specified by the developers.

## 6 Limitations

The current version of mShield has the following limitations, and we conduct the potential solutions in this section.

- Multi-thread support. In mShield, the invoker and the TPL could be multi-threaded, but need to have only one thread when the invoker calls any TPL function and the TPL function returns. This limitation exists, because the current version of mShield encrypts the heap, stack and global variables before any invocation of TPL functions, and performs the decryption once the TPL function returns. When the invoker calls multiple TPL functions or obtains the TPL processing results through multiple threads simultaneously, the invoker will fail to execute after one invocation of TPL functions as the memory is encrypted, and performs the decryption on one TPL function returns which allow the TPL's code read the plaintext memory. To mitigate this limitation, we could borrow Wedge [3] to split the multi-thread application into multi-process one, or use locks to ensure only one thread during the invocation of TPL functions.
- Callback function support. The callback function allows the TPL to call the invoker's function (i.e., the callback function). As `mShield_decrypt` has not been executed, the invoker's memory remains encrypted, and the callback function cannot be executed normally. To mitigate this limitation, we provide `mShield_declare_ltm` and `mShield_declare` to exclude the memory used by the callback function to ensure it can be executed normally. Also, the developer could invoke `mShield_decrypt` at the beginning of the callback function to ensure that the callback function (provided by the invoker) could be executed normally, and call `mShield_encrypt` at the end of the callback function, making the invoker's memory remain encrypted once the callback function returns.

## 7 Conclusion

This paper provides mShield, an SGX-based user-mode solution to protect applications from (unknown) vulnerable or malicious TPLs. With mShield deployed,

the adversary fails to read the application's sensitive memory via the vulnerabilities of the integrated TPLs, even the vulnerabilities are found after the release of the applications. Moreover, mShield could be deployed on COTS platforms directly, as it works in the user mode, only needs the widely deployed Intel SGX, and requires no modification on the OS, hypervisor nor runtime systems. Also, mShield is easy to integrate for the developers who only need to invoke at most 8 functions of mShield as a normal library. We implement a prototype of mShield. The experiments demonstrate that mShield effectively protects the invoker from the in-process memory disclosure attacks exploiting vulnerable or malicious TPLs, and the introduced performance overhead is modest.

## References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS), pp. 340–353 (2005)
2. Arm: Memory domains. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html
3. Bittau, A., Marchenko, P., Handley, M., Karp, B.: Wedge: splitting applications into reduced-privilege compartments. In: 5th USENIX Symposium on Networked Systems Design & Implementation (NSDI). USENIX Association (2008)
4. Brumley, D., Song, D.X.: Privtrans: automatically partitioning programs for privilege separation. In: Proceedings of the 13th USENIX Security Symposium, pp. 57–72 (2004)
5. Cao, C., et al.: CryptMe: data leakage prevention for unmodified programs on ARM devices. In: Bailey, M., Holz, T., Stamatogiannakis, M., Ioannidis, S. (eds.) RAID 2018. LNCS, vol. 11050, pp. 380–400. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00470-5_18
6. Carr, S.A., Payer, M.: DataShield: configurable data confidentiality and integrity. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS), pp. 193–204 (2017)
7. Chen, X.X., et al.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. ACM Sigplan Not. **43**(3), 2–13 (2008)
8. Chen, X., Dick, R.P., Choudhary, A.N.: Operating system controlled processor-memory bus encryption. In: Design, Automation and Test in Europe (DATE), pp. 1154–1159 (2008)
9. Chen, Y., Reymondjohnson, S., Sun, Z., Lu, L.: Shreds: fine-grained execution units with private memory. In: IEEE Symposium on Security and Privacy (IEEE S&P), pp. 56–71 (2016)
10. Colp, P., et al.: Protecting data on smartphones and tablets from memory attacks. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 177–189 (2015)
11. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptology ePrint Archive 2016/086, pp. 1–118 (2016)
12. Docker Inc.: Docker overview (2018). https://docs.docker.com/engine/docker-overview/
13. Duc, G., Keryell, R.: CryptoPage: an efficient secure architecture with memory encryption, integrity and information leakage protection. In: 22nd Annual Computer Security Applications Conference (ACSAC), pp. 483–492 (2006)

14. Durumeric, Z., et al.: The matter of heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference (IMC), pp. 475–488 (2014)
15. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over ASLR: attacking branch predictors to bypass ASLR. In: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, p. 40 (2016)
16. Frassetto, T., Jauernig, P., Liebchen, C., Sadeghi, A.: IMIX: in-process memory isolation extension. In: 27th USENIX Security Symposium, USENIX Security, pp. 83–97 (2018)
17. Götzfried, J., Dorr, N., Palutke, R., Müller, T.: HyperCrypt: hypervisor-based encryption of kernel and user space. In: 11th International Conference on Availability, Reliability and Security (ARES), pp. 79–87 (2016)
18. Götzfried, J., Müller, T., Drescher, G., Nürnberger, S., Backes, M.: RamCrypt: kernel-based address space encryption for user-mode processes. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS), pp. 919–924 (2016)
19. Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: 2015 IEEE Symposium on Security and Privacy (IEEE S&P), pp. 3–19 (2015)
20. Halderman, J.A., et al.: Lest we remember: cold-boot attacks on encryption keys. Commun. ACM **52**(5), 91–98 (2009)
21. Henson, M., Taylor, S.: Beyond full disk encryption: protection on security-enhanced commodity processors. In: Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS, vol. 7954, pp. 307–321. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38980-1_19
22. Horsch, J., Huber, M., Wessel, S.: TransCrypt: transparent main memory encryption using a minimal ARM hypervisor. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp. 152–161 (2017)
23. Kilpatrick, D.: Privman: a library for partitioning applications. In: Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference, pp. 273–284 (2003)
24. Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., Song, D.: Code-pointer integrity. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 147–163 (2014)
25. Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., Druschel, P.: Light-weight contexts: an OS abstraction for safety and performance. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 49–64 (2016)
26. Liu, Y., Zhou, T., Chen, K., Chen, H., Xia, Y.: Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1607–1619 (2015)
27. Mashtizadeh, A., Bittau, A., Boneh, D., Mazieres, D.: CCFI: cryptographically enforced control flow integrity. In: 22nd ACM Conference on Computer and Communications Security (CCS), pp. 941–951 (2015)
28. McKeen, F., et al.: Innovative instructions and software model for isolated execution. In: 2nd Workshop on Hardware and Architectural Support for Security and Privacy (HASP), p. 10 (2013)
29. Microsoft: Data execution prevention (DEP) (2006). http://support.microsoft.com/kb/875352/EN-US/
30. Peterson, P.: Cryptkeeper: improving security with encrypted RAM. In: IEEE International Conference on Technologies for Homeland Security (2010)

31. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: Proceedings of the 12th USENIX Security Symposium (2003)
32. Stewin, P., Bystrov, I.: Understanding DMA malware. In: Flegel, U., Markatos, E., Robertson, W. (eds.) DIMVA 2012. LNCS, vol. 7591, pp. 21–41. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37300-8_2
33. Sun, M., Tan, G.: NativeGuard: protecting android applications from third-party native libraries. In: Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks, pp. 165–176 (2014)
34. veracode: state of software security 2016. Technical report (2017)
35. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: Proceedings of the 22th USENIX Security Symposium, pp. 337–352 (2013)
36. Zhou, Y., Wang, X., Chen, Y., Wang, Z.: ARMlock: hardware-based fault isolation for ARM. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 558–569 (2014)