SuperCall: A Secure Interface for Isolated Execution Environment to Dynamically Use External Services

Yueqiang Cheng[1(✉)], Qing Li[2], Miao Yu[1], Xuhua Ding[3], and Qingni Shen[2]

[1] CyLab, Carnegie Mellon University, Pittsburgh, USA
{yueqiang,miaoy1}@andrew.cmu.edu

[2] Department of Information Security, School of Software and Electronics, Peking University, Beijing, China
qingli@pku.edu.cn, qingnishen@ss.pku.edu.cn

[3] School of Information Systems, Singapore Management University, Singapore, Singapore
xhding@smu.edu.sg

Abstract. Recent years have seen many virtualization-based Isolated Execution Environments (IEE) proposed in the literature to protect a Piece of Application Logic (PAL) against attacks from an untrusted guest kernel. A prerequisite of these IEE system is that the PAL is small and self-contained. Therefore, a PAL is deprived of channels to interact with the external execution environment including the kernel and application libraries. As a result, the PAL can only perform limited tasks such as memory-resident computation with inflexible utilization of system resources. To protect more sophisticated tasks, the application developer has to segment it into numerous PALs satisfying the IEE prerequisite, which inevitably lead to development inefficiency and more erroneous code. In this paper, we propose SuperCall, a new function call interface for a PAL to safely and efficiently call *external* untrusted code in both the kernel and user spaces. It not only allows flexible interactions between a PAL and untrusted environments, but also improved the utilization of resources, without compromising the security of the PAL. We have implemented SuperCall on top of a tiny hypervisor. To demonstrate and evaluate SuperCall, we use it to build a PAL as part of a password checking program. The experiment results show that SuperCall improves the development efficiency and incurs insignificant performance overhead.

1 Introduction

Numerous Isolated Execution Environments (IEE) [4,8,11,17,20,23] have been proposed using virtualization techniques to tackle attacks from both the user and kernel spaces. An IEE separates a Piece of Application Logic (PAL)'s execution from the rest of the platform, including the operating system, so as to protect its execution integrity as well as data secrecy. An indispensable prerequisite of

B. Thuraisingham et al. (Eds.): SecureComm 2015, LNICST 164, pp. 193–211, 2015.
DOI: 10.1007/978-3-319-28865-9_11

an IEE's protection over a PAL is the *self-contained* property stating that the execution flow does not leave the PAL's code, which implies no function calls to external code including the kernel code. The reason of this restriction is that sharing the same execution flow jeopardizes the security of the PAL and IEE.

One limitation of PAL-based IEE is its impact on the utilization of system resources. Without dynamically resource (e.g., memory) allocation and deallocation services, the PAL has to acquire all needed resources before execution and hold them till the end. For instance, the OS allocates to the PAL a bulky memory region with the maximum size in estimation. Such resource usage strategy is obviously not efficient. Moreover, the self-contained property requires the PAL to assemble all needed inputs before its execution. Dynamic data or events generated at runtime cannot be used as an input, which significantly limits the PAL's functionality. At last, the PAL based IEE also introduces great efforts into the development of PAL. For instance, developers have to carefully write their PAL code to avoid invoking library function calls. As a result, the PAL capable of running within an IEE is either small with limited functionality (e.g., computation only), or cumbersome with a higher chance of harboring vulnerabilities.

In this work, we propose a novel interface for a PAL inside an IEE to safely utilize external functions and system calls, e.g., to allocate/deallocate memory buffers and to load encrypted files. It dismisses the self-contained PAL prerequisite and allows the PAL developers to code PALs like a normal program. Our new mechanism is called *SuperCall* as depicted in Figure 1. SuperCall separates the execution flow of the external code from the isolated one, and ensures that the invocation and return procedures always go through the predefined out-and-back gates. Out gates are for safely switching isolation spaces (i.e., from IEE space to non-IEE space) and facilitating call invocations, and back gates are for securely resuming IEE's execution flow, e.g., restoring execution context as well as sanitizing and validating inputs (i.e., return values). Due to the non-bypassable verifications in the back gates and the secure space switches, SuperCall is able to defend against Iago attacks [5] and code reuse attacks [3], and keep other desired security properties, e.g., code and data integrity, data secrecy and control flow integrity.

SuperCall is a new interface, allowing existing IEEs to actively and securely invoke external services. It is similar to upcall [16] that allows the hypervisor to actively invoke guest services. Due to SuperCall needs to validate all inputs of the back gates, developers should carefully select the external services to minimize the validation costs. If the cost is quite high, e.g., requiring numerous code or a long execution, we suggest to reconsider about the possibility of adding the external service into the PAL. There are two typical scenarios for SuperCall. One is to dynamically update resources (e.g., memory can be dynamically allocated/released by malloc-like functions), providing flexible usage model, instead of preserving them in a maximum estimation. The second one is to do securely data exchange with untrusted environments, e.g., saving or reading encrypted files. To demonstrate these two typical scenarios, we have implemented a

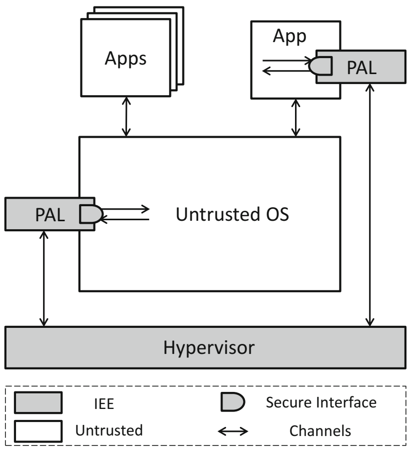


Fig. 1. PALs within IEEs on virtualization-based system. PALs can securely communicate with untrusted applications/OS functions via the SuperCall interface.

password authentication scheme called PwdChecker which is a secure login console in a multi-user system and uses passwords to authenticate users with secret questions as a back-up means. It uses SuperCall to dynamically request memory, load an encrypted database (i.e., secure questions and answers) and get user secure answers. Moreover, this case study also demonstrates that the development efforts of PAL are much reduced. We have conducted performance evaluation of SuperCall by using micro-benchmark tools. The results indicate that the performance overhead of SuperCall is reasonably small.

Organization In the next section, we explain the background and the setting of the problem we undertake to resolve, and present the overall design of SuperCall. Then we describe the typical execution flow and a SuperCall and present the typical application scenarios in Section 3 and Section 4. In Section 5, we use a case to demonstrate the benefits of using SuperCall, and further evaluate the incurred cost. We discuss the related work in Section 6. Section 7 concludes the paper.

2 The Problem Definition and Design Overall

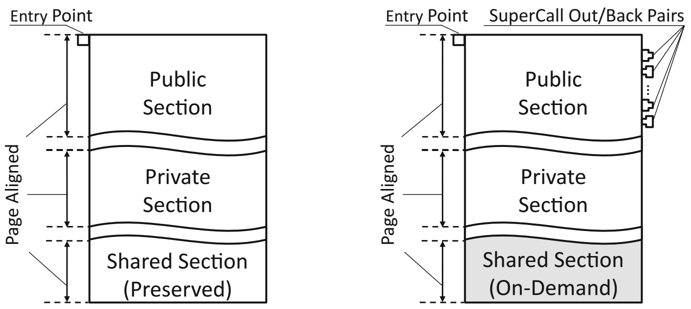
In this section, we first explain the background of the PAL in existing literature, and then highlight our goals followed by a description of the security assumptions. At last, we generally describe how a SuperCall works.

2.1 Piece of Application Logic (PAL)

As shown in [23], PALs in various isolation systems share a common layout consisting of three sections depicted in Figure 2(a). The *private* section contains the security related data, such as cryptographic keys, and other sensitive information, such as credit card numbers. Accesses to the private section are only allowed if they are from the PAL. Any external access is blocked. Note that the PAL's stack and heap regions are also in this section and they are not shared with untrusted code.

The *public* section contains *read-only* information shared between the PAL and untrusted code. It contains the PAL's code and constant data, such as constant numbers and strings. The public section defines the entry point address for the PAL to start execution. Any execution flow not originating from the entry point is not allowed by the IEE, so as to prevent ROP [3] like attacks whereby the adversary twists the control flow to a chosen instruction for a malevolent purpose.

The *shared* section is for data exchange between the PAL and the external environment. Although it is writable for both of them, their accesses are exclusive to each other. This is to deal with the Time-Of-Check-To-Time-Of-Use (TOCTTOU) attack which alters the data when the PAL is about to use sanitized data in the section. Note that this section could be dynamically allocated at runtime (Figure 2(b)).



(a) The layout of traditional PAL.

(b) The layout of PAL with SuperCall support.

Fig. 2. The layout of PAL. The shared section (shaded region) could be dynamically allocated for PALs with SuperCall support.

2.2 Desired Security Properties

We consider a PAL under the protection of the SuperCall. Specifically, the hypervisor maintains a table for a pre-registered PAL's section information as well as the entry points. It checks the initial integrity of the PAL's public and private sections and ensures that their memory pages are exclusively occupied by the PAL. Upon this, SuperCall is able to ensure address space isolation and execution integrity of a PAL.

Address Space Isolation. Address space isolation implies code integrity, data integrity and secrecy. Various isolation mechanisms have been proposed [8,11,17,23] which leverage the processor virtualization to prevent illicit software accesses. Specifically, the hypervisor controls the attribute bits of the Extended Page Table (EPT)[1] entries to specify the desired access control permissions according to the entity occupying the CPU. To prevent malware from making DMA access to unauthorized memory regions, the hypervisor leverages device virtualization to block illicit DMA accesses by configuring the IOMMU page table in the same way as the EPT entries. Note that since all memory resources are allocated before PAL execution, the address mapping is never changed during PAL execution. Thus, the hypervisor can enforce the isolation in the beginning of the PAL and freeze the address mapping until the PAL exits.

Execution Integrity. Execution integrity refers to the property that PAL actually executes with inputs *P_ins* and produces outputs *P_outs*. It implies control flow integrity (CFI), code and data integrity. The hypervisor enforces that the execution flow of PAL always starts to run from a pre-defined entry point, e.g., a back gate or the entry point. At runtime, hypervisor isolates the entire execution environment of the PAL from the rest of the platform without allowing any intervention, so that the PAL's context and control flow are not exposed to any untrusted code.

2.3 Design Goals

We aim to design the SuperCall mechanism for the PAL to securely call external (untrusted) code without undermining the aforementioned security properties. Through SuperCall, a PAL can efficiently invoke system calls and library functions, e.g., invoking *malloc* to allocate memory buffers or issuing *mmap2* for a file reading.

To make SuperCall secure, efficient and practical, we use the following criteria to guide our design.

- **Small TCB.** The TCB of SuperCall should be small and simple. It minimizes the risk of subverting the TCB and allow for formal verification [12]. This property implies that the size expansion and complexity increasing of the hypervisor should be minimum.
- **High Efficiency.** The SuperCall interface should have minimum performance impact on the PAL execution, the IEE protection and the platform as a whole. In addition, SuperCall should minimize the latency for one invocation by reducing unnecessary operations and simplifying interactions.
- **Easy to Use.** The APIs of SuperCall should be easy to use. Thus, the calling convention of SuperCall is the same as regular system calls. SuperCall provides a routine as a wrapper to handle the minor differences which is therefore transparent for PAL developers.

[1] In AMD's virtualization terminology, the Nested Page Table plays the same role as Intel's EPT.

– **Well Defined Entry Points.** The entry point for the SuperCall should be well defined, and the inputs of each entry point should be sanitized and validated before using them, which aim to defend against Iago attack [5].

2.4 Assumptions

We consider a subverted commodity OS as the adversary. This is a realistic threat, since the legacy OS usually has a large code base and a broad attack surface. After gaining the root privilege, the adversary can launch arbitrary code and DMA operations to access or even modify any memory regions and other system resources, e.g., Model Specific Registers (MSRs). The purpose of the adversary is to compromise the security properties of the PAL, for example, to tamper with the PAL's private data and/or to manipulate it execution logic. SuperCall requires that the underlying platform supports hardware-assisted virtualization techniques, and the hypervisor is trusted. We also assume all the I/O devices are trusted and always behave according to their hardware specifications. In this paper, we do not consider attacks that involve physical control of the platform. In addition, we do not consider side channel attacks.

2.5 Overview of SuperCall

The semantics of SuperCall is the same as a function call as shown in Figure 3 where the control flow transfers from the caller (X) to the callee (Y), and returns back after the end of the execution of Y. During the development of a PAL, people could simply replace the function with a SuperCall and add the corresponding out and back gates/interfaces for parameter marshaling and inputs validation (e.g., defending against Iago attack [5]).

Specifically, when caller X attempts to invoke callee Y through a SuperCall, the PAL firstly transfers the control flow to the corresponding *out gate*. The out gate prepares the stack frame needed by the called function (Y) and do the parameter marshalling. The hypervisor saves the context of PAL, and isolates the PAL by manipulating the guest context and transfers the execution flow to the callee function. In SuperCall, this request is issued through a dedicated hypercall, named as *SuperEnter*. When returns, the callee function[2] issues another hypercall, *SuperExit*, to notify the hypervisor return to the corresponding *back gate*. In the back gate, all returns should be sanitized and validated before they are used. The SuperEnter and SuperExit together indicate the start and the end of a SuperCall. Their working style is similar to fast-system-call instruction pair SYSENTER and SYSEXIT [14].

[2] In the implementation, an inserted code issues the SuperExit hypercall for the callee function (details in Figure 4).

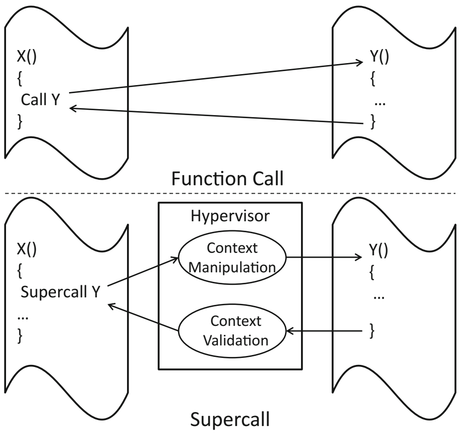


Fig. 3. The SuperCall mechanism. a SuperCall is quite similar to a traditional function call, but it always go through well-defined interfaces and invoke the hypervisor to protect the control flow transitions.

3 Typical Control Flow of SuperCall

A typical control flow of a SuperCall is also similar but relatively complex comparing to the control flow of the traditional function call. It always starts from an out gate and ends with a back gate, involving two space switches driven by a SuperEnter and a SuperExit respectively, as despited in Figure 4.

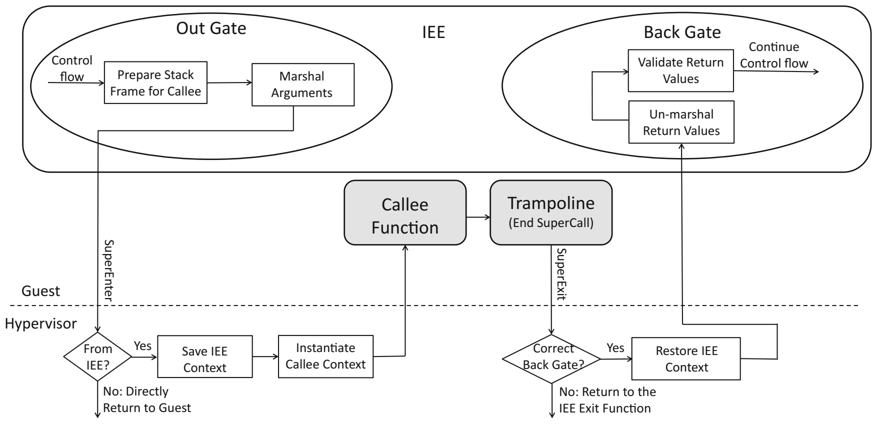


Fig. 4. The execution path of the SuperCall interface. The shaded operations are executed in the untrusted guest environment. Other operations are trusted and executed either in the PAL or in the hypervisor.

3.1 Out Gate

The out gate that is like a wrapper of the callee function shares the same calling conversion and the same parameters with the callee function. After doing several pre-processing operations, the out gate would transfer the control flow to the callee function. Specifically, it does two main tasks: 1) prepare a stack frame for the callee function, and 2) do argument marshalling. It is relatively easy to finish the first task, as the out gate could reuse the stack frame prepared by its caller. The only update is for the return address, which should point to the entry of the prepared trampoline (Figure 4). During the argument marshalling, all non-pointer arguments are kept the same in the stack frame. For pointer arguments, the out gate will move the pointed data, e.g., structures or buffers, into the shared section, and update the pointers to point to the new copies. After these two tasks, the out gate will issue a SuperEnter to inform the hypervisor to transfer the control flow to the callee function.

3.2 SuperEnter

The SuperEnter has two main purposes: 1) functional purpose which aims to achieve the control flow transferring like the traditional function call (i.e., transferring the control flow from the caller to the callee), and 2) security purpose that aims to keep the desired security properties of PALs.

Functional Requirement. SuperCall should be able to transfer the execution flow to the callee function, and let the callee function execute as normal. To achieve these, the following information should be provided and set properly:

- The arguments needed by the callee function. They are necessary for the execution of the callee function.
- The starting address of the callee function. SuperEnter requires it to continue the execution flow from an address specified by the caller function, and that address can not be calculated in advance.
- The stack used by the callee function. Its stack should be different from the one used by the PAL due to the security requirement.
- The return address: The callee will return to this address to indicate the end of its execution flow.
- The entry point of a back gate: The control flow of the PAL will restart from the specific back gate.

All stack-based arguments are handled by the out gate. Thus, the SuperEnter only ask the hypervisor to handle the arguments that are passed through registers. Note that it is not safe that the out gate in the PAL space to directly set those registers, because the values in some or all of them would be flushed or replaced by the hypervisor during its execution for serving the *SuperEnter* request.

Besides these arguments, the SuperEnter has to prepare some additional information to smoothly transfer the control flow. In particular, the entry of the callee function and the callee's stack should be provided. In addition, to make the control flow correctly resume, the entry of the corresponding back gate should be also specified. The provided information as well as the identity of the PAL is safely saved in a dedicated list within the hypervisor space, which is always inaccessible for the untrusted execution environment. Note that the saved record will be used for the validation of the return flow, as well as the context restoration.

Context Manipulation. After getting such information, the hypervisor needs to manipulate the context to let the callee function execute as normal. The hypervisor achieves it by leveraging the processor virtualization technique. Specifically, in hardware-assisted virtualization, almost all guest context information is automatically stored in a dedicated control structure, named as the Virtual Machine Control Structure (VMCS) in Intel VT-x [14]. Only the general registers are manually saved by the hypervisor. The hypervisor is able to read and write the VMCS and the saved general registers. Thus, it manipulates the values of the corresponding registers before the processor enters the guest domain using VM-entry instructions (i.e., $VMLAUNCH$ and $VMRESUME$). More specifically, the hypervisor can modify the IP value to let the guest start the execution from the called starting address, and change the stack pointer SP to assign the top of the stack for the callee function.

Security Requirement. The two basic security properties (i.e., address space isolation and execution integrity) of the PAL should be guaranteed by the hypervisor during the SuperCall process. Specifically, the hypervisor should protect both memory regions occupied by the PAL and the context registers temporally used by the PAL. The private data and all code of the PAL are located in the PAL memory regions. Any malicious modifications and/or illicit reads are possibly lead to the integrity breaking and/or the leakage of the sensitive information. Even worse, the modifications of control data (e.g., function pointer) will subvert the control flow integrity. The context registers can contain temporal data relevant to the private data or even the cryptographic keys. Moreover, a smarter attacker is able to infer more sensitive information from the leaked seed-data. Some registers can also impose the execution behaviors of the PAL, e.g., if the stack pointer SP is illicitly modified, the PAL will fetch wrong local variables or even use incorrect return address, violating the control flow integrity.

To protect the memory regions, the hypervisor could prepare two EPTs/NPTs. One EPT_{iee} is for the PAL, where the memory regions occupied by the PAL are accessible, and another one EPT_{others} is for the untrusted code, where all PAL memory regions are completely inaccessible. When the PAL occupies the CPU, the hypervisor installs the EPT_{iee}. When the PAL invokes the SuperCall to transfer the control flow to an external function, the hypervisor switches to another one EPT_{others}. In this way, the untrusted code occupying the CPU still can not access the memory regions of the PAL.

To protect the context information in the processor registers, the hypervisor should save them before passing the control to the untrusted code, and restore them after the execution flow returns. Specifically, the guest context includes the general registers, flag registers, stack pointer, instruction pointer, as well as the segment descriptors and selectors. After the backup, the hypervisor clears the values of general registers, flags register, stack pointer, instruction pointer and segment descriptors/selectors to avoid the potential data leakage. Note that the stack pointer, instruction pointer and/or general registers will be set for the callee function according to the request of the caller. Note that the SuperCall states are maintained in PAL granularity and saved separately. Thus, those states would not intervene with each other.

3.3 SuperExit

Similar to SuperEnter, the functionalities of SuperExit are also separated into functional and security aspects. For the functional support, SuperExit is to inform the exit of the previous SuperCall. Specifically, the caller function prepares the return address for the callee function. When the callee function finishes and returns, the processor automatically loads the prepared return address into instruction pointer (IP), and jumps to the specific address to continue the execution. The specific address is a prepared trampoline, which can be located in the PAL public section or a pre-defined address (similar to the *vdso* on Linux platform for assisting system calls [19]). Note that the data in the Private section, before the SuperCall returns, is inaccessible. In order to allow the PAL to perform full operations (e.g., accessing the private data), the SuperExit is non-bypassable. This exit request is sent through a dedicated hypercall - SuperExit. More specifically, we put a SuperExit at the very beginning of the prepared trampoline, which is able to guarantee that the processor immediately perform the $VMCALL$ instruction.

Context Validation. The primary purpose of the context validation is to guarantee that the resumed control flow is correct. Recall that the hypervisor saves the PAL and the entry of the back gate during the invocation of the SuperEnter. Thus, the hypervisor will attempt to locate the record by searching in the saved items. If there is one record matched, the hypervisor will remove the record and close this temporal entry point from this PAL before returning to the PAL. Otherwise, there must be something wrong in the current guest execution flow. For such cases, the hypervisor can either inform the guest using an error code or directly terminate the PAL with a fatal error.

The last task of the context validation is to restore the original context of PAL. The unrelated registers, such as ESP and EBP, will be overwritten by previously saved PAL's context. If the hypervisor misses this step and directly reuse the untrusted context left by the SuperExit, the execution integrity is likely to be broken. Note that the hypervisor needs switching back to EPT_{iee} to allow the PAL to access its private data. At the same time, the hypervisor has to restore the context of the PAL, as the context used by the SuperExit is not trusted. If the hypervisor reuse the context left by the SuperExit, the execution integrity is likely to be broken, e.g., the start point of the resumed control flow could be wrong.

3.4 Back Gate

When the callee function finishes and the execution flow returns to the PAL, a *SuperExit* is immediately issued to indicate the end of the SuperCall. From that time, the hypervisor isolates the PAL and restores the control flow. The following work for the *back gate* is to un-marshal and validate the return values, and continue the original execution flow. Unmarshalling return values is the reverse operation of the parameter marshaling. If the return values are non-pointer values, PAL could directly use them. If the return values are pointer values, the back gate should move the pointed data into the private section, and update the pointer accordingly.

After the unmarshalling of return values, the control flow will move to the return validation procedure. In our SuperCall design, each back gate has its own validation procedure, and guarantees that the control flow always goes through the validation procedure before resuming the original control flow. The validation code for a specific input is usually small and simple. Thus, the IEE developers could manually verify its correctness. In addition, it is highly possible to formally verified using certain formal verification methods [12]. As all inputs of back gates are sanitized and validated, an adversary cannot bypass the verification to launch Iago attacks [5] or code reuse attacks.

4 Typical Scenarios

There are two main scenarios for PALs to invoke external functions: 1) update (i.e., allocate/release) memory resources (e.g., main memory and I/O ports), such as allocating/deallocating memory, and 2) exchange data with outside, such as getting file/socket content, reading the inputs of peripheral devices (e.g., user passwords, biometric information), or processing data instead of PALs (e.g., sorting data).

4.1 Resource Update

The first typical scenario is to update memory resource. In the real cases, a PAL usually need extra memory for new inputs or generated data. For a PAL without SuperCall support, it has to allocate a bulky memory with the maximum size in estimation. Obviously the resource usage is not efficient in this situation. With SuperCall support, a PAL does not need to do pre-allocation, instead it could dynamically allocate memory according to the real demand.

To securely use the dynamically allocated memory, the PAL has to trace the memory boundary and requires the protection from the hypervisor. Specifically, in the validation step of the back gate, the PAL gets the boundary (i.e., the start address and the length) of the newly allocated memory resource. Once it is done, the PAL issues a hypercall to the hypervisor to mark the occupied physical memory into its own address space. If the memory is PAL's private resource, it will be marked into the private section. If it is for sharing with others, it will be put into the shared region. Once the newly allocated memory resource is moved into the PAL's address space, the hypervisor will set proper access permissions to grant legal PAL accesses and prevent illicit accesses that are originated from outside of the PAL.

When the allocated memory is not needed, the PAL will release it for maximizing resource utilization. In such cases, the PAL needs to inform the hypervisor to remove the resource from its address space. In particular, the out gate of the PAL collects the memory boundary of the memory resource and issues a hypercall to inform hypervisor that it does not exclusively occupy this memory resource. Note that there would be some problems if the releasing notification to the hypervisor is done in the back gate, because the released memory could be immediately reused by others, before the execution flow returns. In this case, there will be exception to indicate the access violation. If it happens in the user space, the corresponding process would be killed. If it is in the kernel space, it could lead to unrecoverable events, such as system shutdown or rebooting.

4.2 Data Exchange with Outside

Another typical scenario of using SuperCall is to exchange data. In the real cases, a PAL usually needs to exchange data with outside, such as sending dynamic output data (e.g., log file, warning messages) or receiving dynamic input data (e.g., user name, password and PIN number). In the PAL without SuperCall support, it has to get all possible inputs at the very beginning and send all output data at the final end. In this case, the developers have to predict all possible inputs needed by the PAL. In certain extreme cases, the number of the possible combination is extremely large, and consequently the needed memory region is huge. In addition, the generated output data could also occupy a large number of memory regions. If one of these two extreme cases happens, the PAL has to be totally redesigned or divided into many smaller pieces. All these cases imply the impracticality and inflexibility of the traditional PAL design. With the help of SuperCall, the PAL could dynamically receive the inputs according to the real demand. In addition, the PAL also does not need to hold the generated output data to the end of the whole control flow, it could send output data as normal.

To securely send the output data, the PAL has to prepare the data in its private section or the shared section. For the common cases, the out gate could handle them automatically, such as sending a string message or a binary stream. For certain cases, the related data structures could be extremely complex. Facing such conditions, SuperCall has to rely on the developers to manually handle those output data. To facilitate the implementation of SuperCall and allow SuperCall to automatically handle the output data, we recommend that all output data are processed into a string or a binary stream.

In addition, all I/O data could be selectively encrypted according to the requirement of the PAL, achieving secure I/O. The encryption key will be provided by the hypervisor. As the hypervisor space is inaccessible for the untrusted guest environment, the key generation process done by the hypervisor is secure. The encrypted data could be safely stored in the hard drive or send to the remote cloud server.

5 Evaluation

To evaluate SuperCall, we have implemented an exemplary application to use SuperCall, and then measured its performance using several benchmark tools.

5.1 Case Study: PwdChecker

To conduct the case study, we develop an application called PwdChecker which performs the back-end authentication of a remote server. The logic of PwdChecker is as follows. It first loads user password file and the secure question database from the disk to the main memory. It then accepts user inputs including user name and password. If the password is incorrect, PwdChecker allows the user to have another try and increases the login-attempt counter accordingly. When the counter is more than three, it challenges the user with a predefined question. The user has the last chance to get authenticated by supplying the correct answers.

The details of the workflow is depicted in Figure 5 which shows runtime inputs are fed in different stages. There are three dynamic inputs and one static inputs. The static inputs are the inputs passed as parameters (i.e., username and password), while the dynamic inputs are dynamically got at runtime according to

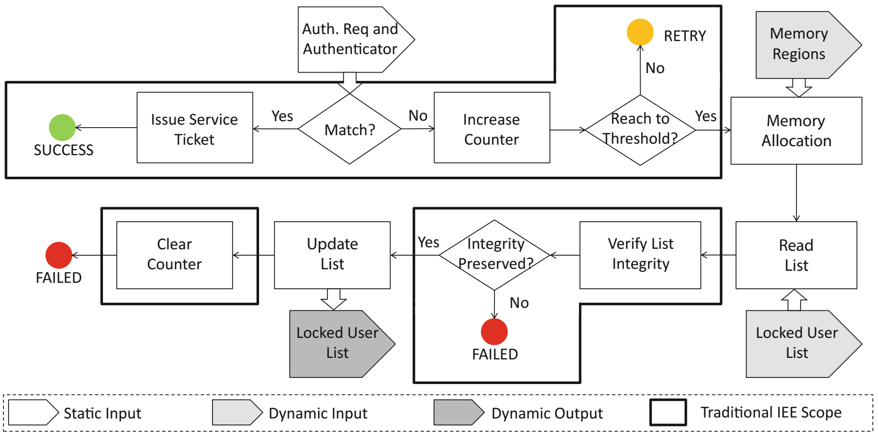


Fig. 5. The work flow of PwdChecker.

the demands (e.g., user answers). It is noteworthy that the dynamic inputs could be passed as the static inputs through carefully modifying the code or even the algorithm logic, e.g., the database containing the secure questions and answers can be passed as a static input. However, it will lead to a waste of memory, e.g., the database occupies many memory pages but it may not be used in most cases.

According to this logic, PwdChecker makes at least three types of system calls in order to acquire the needed resources and inputs during runtime.

– Memory allocation. It needs memory buffers to hold data, e.g, the secure questions and answers.
– File operation. It needs to load the database which encloses user authentication related information.
– I/O operation. It needs to read from the device (e.g., a keyboard) the user's inputs, such as user name and passwords.

5.2 PwdChecker without SuperCall

We select the PwdChecker as a representative example to discuss the PAL development, and SuperCall in particular, when considered in comparison with the two alternatives available at current. The first alternative is to put everything inside a single PAL. As a result, it needs great engineering effort to write their own code or customize existing code, e.g., adding a memory management in the PAL. This design will lower PAL's security level because the size of PAL will be dramatically enlarged. The other solution is to separate PwdChecker logic in multiple PALs in order to maintain the self-contain property for each PAL. As shown in Figure 5, PwdChecker is divided into three PALs. With this design, all the three PALs are self-contained and isolated from each other, and the dynamic inputs are now static inputs for each of them. However, this design is likely to introduce the following issues: 1) it breaks the original logic into multiple pieces, which may not be easily divided in most cases; 2) it would lead to a waste of resource, e.g., the PALs will need to reserve the memory with the highest estimation; and 3) it will increase the size and the complexity to manage shared global states and the communication channels.

5.3 PwdChecker with SuperCall

With the support of SuperCall, developers can easily build a PAL with the similar logic to the traditional insecure implementation, as well as the flexible resource utilization. We only describe the additional operations to demonstrate how easy to convert traditional code into self-contained with SuperCall.

The first operation is the stack switch. As introduced in Section 2.1, the stack for the PAL should be separated with the one for the untrusted code. Thus, in the entry point of the PAL, it immediately backups the untrusted stack and switches to its private stack. Before exiting the execution, it switches the stack back to prevent information leakage. To facilitate this step, we introduce two macros with 6 SloC to perform all these backup, switch and restoration operations. The second operation is to prepare stack frame and marshal the arguments for the untrusted callee function. Traditionally, the compiler generates suitable assembly code to implicitly complete these operations. But now we must explicitly do such operations via a dedicated function (i.e., the out gate). The out gate works like a wrapper of the callee function. The caller function firstly invokes the out gate as normal. In the out gate, it copies the arguments into the untrusted stack, and adjusts the top of the untrusted stack. If there are pointers in the arguments, the out gate must copy the content into the shared memory and update the corresponding pointers to keep semantic consistency. SuperCall adds *12 SLOC* to achieve all these goals. The operations in the back gate are case by case due to the return validation processes are different. However, the basic frame is the same. Thus, we insert a framework for each back gate. The left things are to fill the validation operations accordingly. The verification of the encrypted database is to decrypt the ciphered database, re-calculate and compare the hash value with the trusted one. As the memory allocation and deallocation are common in the real cases, we summarize their verification operations into macros. Later, developers could reuse the Macos to further simplify the development.

Although the out and back gates are manually added, we believe that all code could be automatically generated. Even for pointer arguments, it is still possible once the type and the size of the pointed data structure are collected, e.g. from the data structure definitions and/or the runtime parameters.

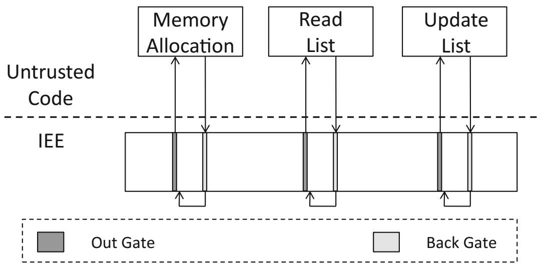


Fig. 6. External function invocations in PwdChecker based on SuperCall.

5.4 Performance Evaluation

We evaluated the performance of our SuperCall implementation and the example app PwdChecker on Ubuntu 10.04 LTS with the Linux kernel 2.6.32.59. These tests were run on a machine with Intel i5-670 CPU (3.47GHZ) and 4GB memory. SuperCall is built upon the Guardian hypervisor [7]. The original Guardian is about $25K$ SLOC, and the SuperCall service adds about 145 SLOC[3].

Firstly, we measure the performance cost of an empty hypercall. It is the baseline to launch a hypercall. This cost can be used later to evaluate the costs of SuperEnter and SuperExit. We create an empty hypercall, and call it from

[3] We use the tool *sloccount* [22] to calculate the source code.

Table 1. The time cost of SuperCall.

Operations	CPU Cycles	Time (μs)
An Empty Hypercall	3879	1.12
SuperEnter	13794	3.98
SuperExit	13438	3.87
SuperCall	27232	7.85

the guest domain. We treat the hypercall as a whole, and measure the round time from issuing the hypercall to its returning (i.e., from guest domain to guest domain via the hypervisor). The time cost (i.e., 1.12μs on average) demonstrate the basic cost of a hypercall. Based on this, we can evaluate the extra cost added in SuperEnter and SuperExit. The measurement results of SuperEnter and SuperExit are listed in Table 1. Because an empty SuperCall contains one SuperEnter and one SuperExit only, the total round-trip time on a SuperCall is about 7.85μs. To further demonstrate the performance cost, we also measure the time cost in the PwdChecker example. The results in Table 2 show that the performance overhead is small that is roughly the cost of one SuperCall. We do not measure the third SuperCall due to the instability of typing answers through keyboard.

Table 2. The measurement results of PwdChecker.

Operations	Time (μs)	Overhead (μs)
Original Malloc	0.08	8.33
Malloc with SuperCall	8.41	
Original LoadDB	30.69	9.45
LoadDB with SuperCall	40.14	

The code expansion is limited due to the support of SuperCall. In the PwdChecker example, there are three out and back gates. All of them together need 180 SLoC in total, which is even far less than the memory allocation function (e.g., `malloc`).

6 Related Work

PAL Protection. There are many existing schemes to protect a PAL [4,17,18,23]. The Flicker [18] system aims to put a PAL into the isolated environment protected by the DRTM technique [9]. Due to the high latency and the poor communication channel, many virtualization-based schemes proposed [4,17,23]. However, in all of them a PAL still has only limited functionalities, without a secure interface to invoke untrusted services. This gap is addressed by our scheme. In addition, many virtualization-based schemes [6,10,13] aim to protect a whole high-insurance application, rather than a PAL. For all of them, the interaction interfaces are system calls that are not well-defined, and therefore surfer from the Iago [5] attack. In our scheme, back gates explicitly sanitize and validate all inputs, with the purpose to defend against Iago attack. The Intel SGX technique [15] and a similar architecture [21] are also promising techniques to protect a PAL or a whole application [2]. Similar to SuperCall, both also require adding specific well-defined interfaces for PALs.

Hypercall. Traditionally, a PAL has only one communication channel, through which the PAL can issue hypercalls to ask for services from the hypervisor. But it now has another new channel, allowing it to communicate with the untrusted code without losing the security properties. The virtualization technique provides hypercall, a communication channel for guest to actively communicate with the hypervisor. In paravirtualization, the hypercall is implemented as an interrupt, e.g., int $0x82$ on Xen [1], similar to the traditional system call mechanism. In the hardware-assisted virtualization, the processor is extended to support a series of virtualization instructions [14], and one of them is to launch a hypercall. In the original design of hypervisor, the return address of a hypercall is always the next instruction of the hypercall instruction. But in SuperCall technique, we reuse the virtualization instructions, and change the return behaviors of hypercalls. Specifically, the SuperEnter returns to the specified callee function, instead of the next instruction. The new return behavior (i.e., SuperExit) is similar to the *SymCall* mechanism [16], but not the same.

Upcall. The SymCall [16] provides a synchronous way (upcall) to invoke a function in a running guest environment. It provides a shared structure between the hypervisor and the guest domain. Through the shared structure, the hypervisor is able to enumerate the available functions (like system calls in syscall table). The guest and the hypervisor can directly read/write to this memory regions without triggering any vm_exit or protection violation. In our SuperCall design, we choose the synchronous way, but do not use the shared structure, because the hypervisor does not need to know the callee functions in advance. Dynamically updating function information to the hypervisor increases the flexibility of SuperCall. The direct benefit is that PAL can freely decide to use which function at runtime, without needing the registration procedure to register to the hypervisor. Another benefit is saving memory and the corresponding maintain cost. If a large number of PALs attempt to use many different functions, the size of the shared structure will be dramatically enlarged in SymCall setting, while in our design, the hypervisor only temporally maintains the function information and throws it away after the end of the execution.

7 Conclusion

In this paper, we introduced SuperCall as a new interface, through which a PAL could securely and efficiently invoke untrusted external functions, increasing the flexibility of interactions and improving the utilization rate of resources. The control flow is escorted by the hypervisor and all inputs of the SuperCall interfaces are sanitized and validated, and therefore Iago attacks and code reuse attacks do not work here. We implemented and evaluated a prototype of SuperCall on Guardian hypervisor by adding 145 SLOC. The experiment results indicated that SuperCall improved the development efficiency with insignificant performance overhead.

Acknowledgments. We are grateful to the anonymous reviewers for their useful comments and suggestions. This research was supported in part by the National Natural Science Foundation of China under Grant No. 61232005.

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP 2003: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 164–177. ACM, New York (2003)
2. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI 2014, pp. 267–283. USENIX Association, Berkeley (2014)
3. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Syverson, P., Jha, S. (eds.), Proceedings of CCS 2008, pp. 27–38. ACM Press, October 2008
4. Champagne, D., Lee, R.B.: Scalable architectural support for trusted software, Bangalore, India, January 9–14, 2010. Nominated for Best Paper Award (2010)
5. Checkoway, S., Shacham, H.: Iago attacks: why the system call api is a bad untrusted rpc interface. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, pp. 253–264. ACM, New York (2013)
6. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.K.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 2–13. ACM, New York (2008)
7. Cheng, Y., Ding, X.: Guardian: hypervisor as security foothold for personal computers. In: Huth, M., Asokan, N., Čapkun, S., Flechais, I., Coles-Kemp, L. (eds.) Trust and Trustworthy Computing. LNCS, vol. 7904, pp. 19–36. Springer, Heidelberg (2013)
8. Cheng, Y., Ding, X., Deng, R.H.: Driverguard: Virtualization-based fine-grained protection on i/o flows. ACM Trans. Inf. Syst. Secur. **16**(2), 6:1–6:30 (2013)

9. INTEL CORPORATION. Intel trusted execution technology (intel txt) c software development guide, December 2009
10. Criswell, J., Dautenhahn, N., Adve, V.: Virtual ghost: protecting applications from hostile operating systems. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, pp. 81–96. ACM, New York (2014)
11. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS 2008, pp. 51–62. ACM, New York (2008)
12. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, pp. 346–355. ACM, New York (2006)
13. Hofmann, O.S., Kim, S., Dunn, A.M., Lee, M.Z., Witchel, E.: Inktag: secure applications on an untrusted operating system. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, pp. 265–278. ACM, New York (2013)
14. Intel. Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c, October 2011
15. Intel. Software guard extensions programming reference, September 2013
16. Lange, J.R., Dinda, P.: Symcall: symbiotic virtualization through vmm-to-guest upcalls. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2011, pp. 193–204. ACM, New York (2011)
17. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: Trustvisor: efficient tcb reduction and attestation. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 143–158. IEEE Computer Society, Washington, DC (2010)
18. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for tcb minimization. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems, Eurosys 2008, pp. 315–328. ACM, New York (2008)
19. nixCraft. Explains: Linux linux-gate.so.1 Library / Dynamic Shared Object [vdso]. http://www.cyberciti.biz/faq/linux-linux-gate-so-1-library-dynamic-shared-object-vdso/
20. Payne, B.D., Carbone, M., Sharif, M., Lee, W.: Lares: an architecture for secure active monitoring using virtualization. In: Proceedings of the 2008 IEEE Symposium on Security and Privacy, SP 2008, pp. 233–247. IEEE Computer Society, Washington, DC (2008)
21. Shinde, S., Tople, S., Kathayat, D., Saxena, P.: PodArch: Protecting Legacy Applications with a Purely Hardware TCB. Technical Report NUS-SL-TR-15-01, School of Computing, National University of Singapore, February 2015
22. Spillner, J.: Sloccount. http://www.dwheeler.com/sloccount/
23. Strackx, R., Piessens, F.: Fides: selectively hardening software application components against kernel-level or process-level malware. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, pp. 2–13. ACM, New York (2012)