Peapods:OS-Independent Memory Confidentiality for Applied Cryptography

LI CONGWU, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
LIN JINGQIANG, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
CAI QUANWEI, State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
LUO BO, Department of Electrical Engineering and Computer Science, the University of Kansas

Cryptographic algorithms are widely used in cryptographic systems and play a significant role.The cryptographic software system is linked to the user's core interests. When users use online transactions, e-mail, and remote login services, they rely on the protection of cryptographic software systems. Once the security of cryptographic software systems is not guaranteed, it will cause extreme problems for users. Big losses; on the other hand, the speed of password computing directly affects the user's experience with the crypto software system. Therefore, it is imperative to provide secure, high-speed and reliable cryptographic services.However, existing programs face various memory disclosure attacks and the security of sensitive data cannot be guaranteed, especially key security.In this paper, We present Peapods, a compiler enhancement tool that provides security enhancements for cryptographic software systems that can defend against memory disclosure attacks.A Peapod represents a protected key-related calculation (especially the private key calculation in the public-key cryptographic algorithm).Through the transactional memory mechanism, we can guarantee that in the course of calculation, once an attacker reads the key, the key will be automatically cleared.If the computing key is in a non-calculated state, it is in a ciphertext state.Further, to ensure that calculations can be completed in the transactional memory protection state, it can be split into multiple partitioned tasks.Between the partitioned tasks will exit the transactional memory state, but the calculated intermediate variables will also be encrypted to avoid the leakage of sensitive information.Our experiments show that there is no problem with security and compared to PolarSSL RSA private-key calculation protected by Peapods and non-protected PolarSSL, the performance loss is only 10% within an acceptable range.

CCS Concepts: •**Computer systems organization** →**Embedded systems;** *Redundancy;* Robotics; •**Networks** →Network reliability;

Additional Key Words and Phrases: Implementation of Cryptographic Algorithm, Software Memory Attack, Cold Boot Attack, Transactional Memory, Sensitive information protection

ACM Reference format:
Li Congwu, LIN Jingqiang, CAI Quanwei, and LUO Bo. 2010. Peapods:OS-Independent Memory Confidentiality for Applied Cryptography. *ACM Trans. Web* 9, 4, Article 39 (March 2010), 13 pages.

This work is supported by XXX.
 1559-1131/2010/3-ART39 $15.00
DOI: 0000001.0000001

DOI: 0000001.0000001

1 INTRODUCTION

Cryptographic algorithms are widely used in cryptographic systems and play a significant role.When we use cryptographic algorithms to implement cryptographic systems, we usually use existing Open Source cryptographic libraries to complete the required calculations and combine them with application functions. The cryptographic software system is linked to the user's core interests. When users use online transactions, e-mail, and remote login services, they rely on the protection of cryptographic software systems. Once the security of cryptographic systems is not guaranteed, it will cause extreme problems for users.On the other hand, the speed of computing directly affects the user's experience with the cryptographic system. Therefore, it is imperative to provide secure, high-speed and reliable cryptographic services. In recent years, many memory protection schemes have been proposed. Copker Mimosa RegRSA . Although these schemes can protect memory information in some scenarios, there are some shortcomings:(1) Only work in kernel-mode. This may have an impact on kernel security and it is difficult to support the existing user-mode cryptographic algorithm library. (2) Only support specific algorithms. Refactoring code will introduce a lot of extra work.Therefore, existing application layer cryptographic software system still face various memory disclosure attacks and the security of sensitive data cannot be guaranteed, especially key security.We need a more principled approach to secure the key in the application layer cryptographic software system.

Peapods

Peapods is a automated compiler enhancement tool that provides security enhancements for application layer cryptographic software systems that can defend against software memory attacks and cold-boot attack.

In our scenario, a Peapod represents a protected key-related calculation (especially the private key calculation in the public-key cryptographic algorithm).Through the transactional memory mechanism, we can guarantee that in the course of calculation, once an attacker reads the key, the key will be automatically cleared.If the computing key is in a non-calculated state, it is in a ciphertext state.Further, to ensure that calculations can be completed in the transactional memory protection state, it can be split into multiple partitioned tasks.Between the partitioned tasks will exit the transactional memory state, but the calculated intermediate variables will also be encrypted to avoid the leakage of sensitive information. Although peapods is mainly dedicated to the protection of keys in cryptographic systems, peapods can also protect user sensitive information when necessary.

Implementation

We implemented Peapods in LLVM for x86 and recompiled PolarSSL and all of its dependencies.We use Peapods to protect Polarssl RSA decryption calculations and users only need to make a few code changes(less than 1000 lines).Experiment shows that compared to PolarSSL protected by Peapods and non-protected PolarSSL, the overhead is only 10% within an acceptable range.

Our contributions

(1) We propose a scheme that can resist memory disclosure attacks in user-mode. (2) Based on LLVM, we implemented Peapods, a tool that automatically implements key protection and it is independent of the cryptographic algorithm and operating system, programmers do not need to refactor the entire cryptographic software, just add a small amount of code,

peapods can provide security enhancements for various cryptographic software systems.The experimental evaluation showed that peapods only introduces a small overhead.

2 BACKGROUND

2.1 Memory Disclosure Attacks

There are many security threats when the cryptographic algorithm is running. One of them is sensitive information stored in the memory, such as a key, which is stolen by an adversary.One of the attacks is a memory disclosure attack. Memory disclosure attacks are read-only memory attacks where the adversary can obtain the contents of the memory but cannot tamper with it. Memory disclosure attacks include software attacks and hardware attacks.Software attack means that the attacker does not physically touch the attack target. It uses software vulnerabilities to bypass the system protection mechanism and illegally accesses the data in the memory.Such as OpenSSL heart bleeding attack, using OpenSSL heartbeat vulnerability, so that remote attackers can obtain a server 64KB memory data by constructing a malicious request.Hardware attack means that an attacker can physically contact an attack target and use a physical method to read the memory of the target.Such as cold-boot attack, using the delay disappearance effect of DRAM, directly read the contents of the memory chip.In short, memory disclosure attacks pose a serious threat to plain-text sensitive data in memory.

physical memory attack

Physical memory attack refers to the behavior of the attacker directly reading the memory chip and obtaining the data illegally after the attacker has physical control over the attack target.In this case, any software-based protection will fail because the attacker can directly manipulate the hardware.Physical memory attack mainly include two types: (1)cold boot attack, which uses DRAM's delay-disappearing effect to directly read the contents of the memory chip, (2)DMA attack, which initiate illegal DMA requests through malicious peripherals, and directly read data from the memory chip.

software memory attack

Software memory attack refers to the behavior of the attacker does not physically contact the attack target, and uses software vulnerabilities to bypass the system protection mechanism and illegally access the data in the memory.For example, OpenSSL heart bleeding attack exploits the vulnerability of OpenSSL heartbeat to enable remote attackers to obtain 64KB of memory data from the server once by constructing a malicious request.Software memory attack mainly include four types: (1)attack based on an isolation mechanism vulnerability, which exploits the defect of the virtual memory subsystem of the operating system to illegally access data from the memory chip, (2)attack based on unclear dynamic memory, which originates from the fact that the dynamic memory storing sensitive information is not cleared and is used by other programs, (3)attack based on cryptographic software's own vulnerability, such as OpenSSL heart bleeding attack, (4)attack based on memory data diffusion, which exploits normal operating system functionality.For example, kernel dumps were originally used to save memory images to disk for developers to debug after a software crash, but attackers can also use this feature to deliberately trigger an exception that causes the software to crash, and then read the sensitive data contained in the process from the disk.

2.2 TSX

Transactional memory is a software technique that simplifies writing concurrent programs.The main idea is to declare a region of code as a transaction. A transaction executes and atomically commits all the results to memory when the transaction succeeds or aborts and cancels all the results if the transaction fails. Transactional memory mechanism provide the Atomicity, Consistency and Isolation qualities. These transactions can safely execute in parallel, and only serial execution is performed in the case of data conflicts, which happen when several threads access the same memory address at the same time and at least one is write operation.When a data conflict occurs, it will cause a rollback and all the results will be canceled.

The recent development is Intels TSX and the implementation in Haswell.Haswell is the first x86 processor to feature hardware transactional memory.Programmers only need to specify critical sections for transactional executions, the processor transparently performs data conflict detection, transaction commit and rollback.TSX tracks the read-set and write-set of a transaction, at a 64B cache line granularity. The read-set and write-set are respectively all the cache lines that the transaction has read from, or written to during execution. A transaction encounters a conflict if a cache line in its read-set is written by another thread, or if a cache line in its write-set is read or written by another thread.

TSX provides two interfaces for programmers to make use of transactional memory. The first mode is Hardware Lock Elision(HLE), provides a pair of compiler hints:xacquire and xrelease to enter or exit the critical section.Once abort happens, the processor will roll back to the original state, and then automatically restarts the execution in a legacy manner. The second mode is Restricted Transactional Memory (RTM), provides three new instructions, XBEGIN, XEND and XABORT, to start, commit, and abort a transactional execution.When using the RTM interface, the programmer needs to specify a fallback handler for the XBEGIN instruction as a parameter and write the fallback handler logic. When a transaction aborts, the program jumps to the specified fallback handler and executes the corresponding program logic.

2.3 LLVM

LLVM is an abbreviation of low level virtual machine. It is actually a compiler framework. With the continuous development of this project, llvm has been unable to fully represent this project, but this name has been continued, and it can now be understood as a full-featured compiler. LLVM can be seen as a collection of compiler and toolchain technologies, and they are modular and reusable.

The traditional compiler is divided into front-end, optimizer, back-end three stages, llvm is also divided into three stages, but slightly different in the design.

Because LLVM is a virtual machine, it has its own Intermediate Representation. It needs to compile the LLVM byte code into a platform-specific assembly language before it can be run by the native assembler and linker to generate assembly code, executable shared libraries, etc.

The front end fetches the source code and then turns it into an intermediate byte code representation.This translation simplifies the work of other parts of the compiler so that they do not need to deal with all the complexity of C++ source code.

In LLVM Optimizer, PASS transforms the program between the intermediate representation.In general, the process is used to optimize the code, that is, the process output program is completely functionally identical to its input program, but the performance is

improved.However, we can also add code logic by adding an IR PASS to achieve a functional improvement over the input program.LLVM provides a lot of APIs for manipulating intermediate bytecode representations, so we can use these interfaces to generate IR directly in memory and run directly to achieve programmatic changes.The intermediate representation of the source code can be further divided into modules, functions, basic blocks, and instruction four-layer structures.

The back-end part translates the intermediate representation into the assembly language of the target platform and generates the actual running machine code.

Our work is to modify the intermediate representation of the source code through a Module PASS during the LLVM optimizer stage, adding logic such as generating a master key, identifying sensitive variables, and encrypting and decrypting sensitive data variables (unlike other situations. At this time, the program output by the process has changed in function compared with the program it has input), which enables the program to automatically protect the identified sensitive data variables and prevent the leakage of sensitive data information.

3 DESIGN

3.1 THREAT MODEL AND SECURITY GOALS

THREAT MODEL:In this paper we assume a powerful attacker who has the ability to read arbitrary areas of memory and overwrite all writable areas of memory.However,as we are concerned with memory disclosure attacks in this paper, the attacker is unable to write to executable memory (marked read-only).Since the registers set to the LLVM compiler reserved registers will be removed from the register allocation pool, the attacker can not read the value of special registers which LLVM compiler reserves either.

Peapods is focused on protecting user-level programs such as PolarSSL RSA decryption calculation, and we do not address protecting the kernel in this paper. We assume that the kernel does not save any user-level registers at least the ones that are used to store the keyduring context switches in user accessible memory. This is true of all major modern operating systems that we are aware of. Custom user-level threading libraries may also require changes to ensure these registers are not saved.

SECURITY GOALS:Under the premise of the above thread model, we proposes Peapods, a LLVM-based compiler enhancement tool that automates the protection of sensitive information in memory to achieve the following security goals:

(1) Refer to the security goals (1), (2), (3) in the Mimosa.
(2) Without using kernel features, peapods automatically protects the key in every key calculation transaction.
(3) Implementation is independent of the cryptographic algorithm and operating system.

The first goal can resist memory disclosure attacks in kernel-mode, the second goal is to provide support for the application layer cryptographic software systems in user mode, the third goal is to enable Peapods to implement security enhancements for a variety of cryptographic software systems.

In summary, Peapods can resist various memory disclosure attacks and provide security enhancements for various application layer cryptographic software systems, automatically.

3.2 Structure of Peapods

Mimosa provides a defense against memory disclosure attacks in kernel-mode.Based on Mimosa, we propose Peapods to defend against memory disclosure attacks in user-mode.Peapods

is a LLVM-based compiler enhancement tool that automatically protects memory that stores sensitive information, programmers do not need to refactor the entire cryptographic software, just add a small amount of code, peapods can provide security enhancements for various cryptographic software systems.We provide users with a new **keyword** to tag sensitive variables, after the user completes the tag, peapods will automatically protect sensitive variables. During Automated process, we need to protect the following three sensitive variables separately:(1) local variable that address was already determined at compile time, (2) global variable or static variable that address was already determined at compile time, (3) variable that address was not determined at compile time.In order to reduce the impact of time interrupts on transactions, we introduced transaction split.Frequent page faults occur during program execution, we designed efficient preload for Peapods to solve page fault problems.

In order to achieve security goals,we divide the operation process of Peapods into three phases:User processing phase, Tool processing phase, Program execution phase

User processing phase:For sensitive variables whose compile-time address has already been determined, the user needs to add the keywords we provide in the place where the sensitive variables are defined. For sensitive variables whose compile-time address is not determined, the user needs to provide information of such sensitive variables,and then they also need to use the interface we provided for parameter passing through specified structure.When it is necessary to assign a value to a sensitive variable, the user needs to call START(args list) before the sensitive variable assignment and END(args list) after the assignment.

Tool processing phase:1.Add initialization logic.2.For sensitive variables whose compile-time address has already been determined, Peapods recognizes variables that have been marked as sensitive data. If it is a local variable, it inserts the XBEGIN instruction before the instruction it defines, and other instructions after the next instruction of its defined instruction, including: (1) Use the master key to perform the CBC mode AES encryption of the intermediate byte code for the variable, and (2) the XEND instruction. Therefore, when these local variables are initialized in the way defined, the sensitive variables are stored in ciphertext in memory. Before the XEND instruction, Peapods will carefully clear the intermediate states used in this stage; If it is a global variable, the AES encryption of the CBC mode of all recognized global variables is completed at the time of program initialization.3.Peapods automatically encrypts and decrypts all sensitive variables in the structure for sensitive variables whose compilation addresses are undetermined.4.START,END, TEM function also adds automatic encryption and decryption protection logic

Program execution phase:We can divide the program execution phase into initialization phase, sensitive variable definition and assignment phase, and protection calculation phase.

We use an AES master key to protect user sensitive data. The AES master key is generated at program startup and is then stored in the LLVM compiler reserved register.Sensitive variables are AES encrypted after user-defined initialization or after assignment. When receiving a request for calculating sensitive information, Sensitive information is dynamically generated, used, and destroyed in transaction execution.When there is no computing task, sensitive information is always stored in ciphertext in memory.

Initialization phase: This stage starts when the program starts, mainly performing the following steps: 1.Randomly generate the AES master key and copy it to the reserved register in each CPU core. 2.Call the main function's preload function. 3.Call memset and other library functions. 4.The global variable IV is randomly generated, and the CBC-mode

AES encryption is performed on all the global variable-sensitive information that has been identified.Finally, the intermediate states used in this phase are carefully cleared.

The first step is to generate the master key; the second and third steps are to prevent Peapods from abort the transaction due to page fault in the code segment; the last step is to store the sensitive information in ciphertext in memory.

Sensitive variable definition and assignment phase: When the user needs to add a sensitive variable definition and assign value to the sensitive variable, Peapods encrypts the sensitive variable in plain text using the AES master key, so that the sensitive variable is stored in memory in ciphertext form.

Protection calculation phase: When Peapods receives a sensitive information calculation request, sensitive information is calculated using the corresponding sensitive information, and then the result is provided to the user.This stage contains the following steps: 1.TSX begins to track memory access in the L1 data cache (maintaining read/write sets). 2.Ciphertext sensitive information is loaded into cache from memory. 3.Master key is loaded into cache from reserved register. 4.Use master key to decrypt sensitive information. 5.Calculation. 6.Clear all sensitive information in the cache and registers except the final calculation. 7.Commit transaction

All memory accesses in Protection calculation phase are monitored by hardware. In particular, we use Intel TSX technology to declare a transaction area in which operations that violate the security principles of Peapods are discovered: (1) Any attempt to access changed memory, because the decrypted plaintext Sensitive information and any private intermediate results are in the TSX write set; (2) Data is synchronized from cache to memory due to cache reclaim or replacement

If the above memory exception does not occur, the entire transaction is committed and the results are returned.Otherwise, the hardware's termination processing logic automatically discards all updated memory and then performs a rollback handler to handle the exception.We will immediately retry sensitive information calculations.

4 IMPLEMENTATION

Our system is a C/C++ compiler built on the Clang/LLVM compiler framework. Any application wishing to be protected by Peapods must be recompiled along with all of its dependencies.

The LLVM module pass provides the sensitive variable identification logic, sensitive information encryption and decryption logic we need.

4.1 Utility functions

We encapsulate the XBEGIN and XEND instructions in user mode into utility functions START and END.They are used to start transactions and end transactions respectively.In addition,We also encapsulate an Utility functions TEM to split transactions.

4.2 Master key protection

The master key needs to be saved in the system for a long time.The master key must be tightly protected, otherwise it will pose a threat to data security throughout the system.As persistent sensitive information, the master key is always stored in secure storage in clear text whether or not the sensitive information calculation is to be performed.For physical memory attacks in memory disclosure attacks, any storage area that can restrict access only at the software level is insecure. Secure storage should have physical mechanisms to restrict

access.Each CPU core or each hardware thread has a separate set of registers that can only be read by the currently executing instruction.Therefore, the register is the storage resource that runs the program exclusive and it is an ideal secure storage. At the same time, XMM registers are mainly used for floating point and vector operations. Excluding a small number of XMM registers does not have a significant impact on performance in non-massive floating point computing environments.Therefore, we decided to store the master key in the XMM register.

The master key is randomly generated by the rdrand instruction at program startup.Then it is stored in the XMM7 register without any memory operations.In the LLVM backend, we set the XMM7 register as a reserved register.At last,user-level threading libraries have also made corresponding changes to prevent the value of XMM7 register from leaking into memory.

4.3 sensitive variables Identification

We divided the sensitive variables into three categories: (1) local variable that address was already determined at compile time, (2) global variable or static variable that address was already determined at compile time, (3) variable that address was not determined at compile time. For sensitive variables whose compile-time addresses have been determined, such as sensitive variables for array types, structure types (members without pointer-type variables), we provide the user with the new **keyword**: attribute ((annotation "Private Key")) when the user defining a sensitive data variable of this type, they need to add the keyword to the variable definition.Peapods will identify the variable that has been marked as sensitive data. If it is a local variable, it inserts the XBEGIN instruction before the instruction it defines, and inserts the instruction following its defined instruction: (1)The intermediate byte code of the logic of use the master key pair to encrypt this variable with AES encryption of the CBC mode, and (2) the XEND instruction.Therefore, when these local variables are initialized in the way defined, the sensitive variables are stored in ciphertext in memory. Before the XEND instruction, Peapods will carefully clear the intermediate states used in this stage;If it is a global variable or static variable, the AES encryption of the CBC mode of all recognized global variables is completed at the time of program initialization.

For sensitive variables with undecided compile-time addresses, such as pointer-type sensitive variables, since the address and length of sensitive variables cannot be obtained at compile time, the user is required to provide information of such sensitive variables. Then, the user uses the specified structure passes the parameters in the utility function provided by us, and Peapods will automatically encrypt and decrypt all the sensitive variables in the structure.

```
1 struct args{
2     unsigned char *data;
3     int len;
4     struct args* next;
5 }*args_list,args_list_next;
```

When users want to assign values to sensitive variables, they need to call START(args_list) before assigning sensitive variables and END(args_list) after assigning values.In IR PASS, we add logic to automatically encrypt and decrypt sensitive variables in utility functions such as START and END. In this way, the entire assignment process is performed under transaction

protection. After the transaction is started, the sensitive variables are decrypted, and then encrypt sensitive variables before ending the transaction. Therefore, there is always only a ciphertext sensitive variable in memory.

It is worth noting that reading and writing files within a transaction will cause the transaction to abort. Therefore, Peapods cannot provide security protection during the assignment phase if the user reads a file to assign sensitive variables.In fact, in this case, the user can still call START(args list) after reading the file, call END(args list) after assignment, and clear the plaintext intermediate variable to make the sensitive variable exist in ciphertext after the assignment.However, in the process of reading a file, sensitive information is exposed in memory in the form of clear text. It is assumed that there is no memory disclosure attack at this time.At the same time, there is a backup of sensitive information in the hard disk. An attacker can also read the hard disk to obtain sensitive information. The user needs to delete the file containing sensitive information after reading the file.

4.4 Sensitive information protection

We need to consider time interruptions during the calculation. The calculation of sensitive information (such as private keys) is usually relatively time-consuming. Therefore, the execution of a transaction that would otherwise be successful may be terminated by a time interruption.In addition, other interrupts can also cause the transaction to terminate.The solution to other kernel-mode memoryless encryption schemes is to disable interrupts,such as TRESOR, PRIME, Copker and Mimosa. We propose transaction split, a method for breaking time-consuming large transactions into multiple small transactions.

The design of the transaction split is mainly to achieve the following goals: 1.Even if the entire time-consuming calculation is not completed, we can still save some time-consuming intermediate calculation results. When the transaction occurs abort, we can use these already calculated intermediate calculation results to start the next calculation. 2.Compared to transactions that were not split prior to the entire calculation, only a small number of CPU clock cycles were consumed in each transaction and only a small amount of memory space was occupied. Therefore, these small transactions are easier to submit successfully.

Since the introduction of transaction splitting, the entire calculation is no longer an atomic operation, we need to encrypt the calculated intermediate calculation results using the AES master key before the transaction ends, and then perform AES decryption after the start of the next transaction to ensure security outside the transaction.These security logic are automatically implemented when the TEM function is called

After we split a PolarSSL RSA decryption calculation into 128 times, the program performs well.

We provide users with three utility functions.We track all function call instructions in IR PASS, recognize the call to START, END, and TEM. Then we insert the corresponding intermediate representation code of encryption and decryption before the END function call instruction, after the START function call instruction, and both before and after the TEM function call instruction respectively.

4.5 Page fault handling

We protected Polarssl RSA decryption calculations with peapods, during the execution of the program, we used perf, a performance analysis tool to help us knowing the program's operating status. Perf relies on the Intel performance monitoring facility, which supports

Table 1. Utility functions list

function name	function	usage method	Implementation logic
START(args_list)	Start a transaction	Used before sensitive operations begin	(1)XBEGIN(2)get the master key from the XMM7 register(3)generate AES context using the master key(4)AES decryption of CBC mode for identified sensitive data variables
END(args_list)	End a transaction	Used after sensitive operations end	(1)AES encryption of CBC mode for identified sensitive data variables(2)Erase intermediate results(3)XEND
TEM(args_list)	Split a transaction	Used in locations where transactions need to be split within the transaction	(1)AES encryption of CBC mode for identified sensitive data variables(2)Generate intermediate variable IV, then randomly initialize and perform AES encryption of CBC mode for the intermediate variables passed in by the user.(3)Erase intermediate result-s(4)XEND(5)XBEGIN(6)get the master key from the XMM7 register(7)generate AES context using the master key(8)AES decryption of CBC mode for identified sensitive data variables

precision-event-based sampling. This function can record the current processor state when a specific event occurs.We monitored the RTM_RETIRED.ABORTED event, which was triggered when RTM execution was terminated.Based on the processor status acquired, we can find out the cause of the termination.Perf reported a lot of the termination of the transaction caused by the pages fault. After analysis and exploration, we found that these pages fault mainly come from pages fault of the code segment.

For the pages fault of the code segment, Peapods implements automatic preloading of the code to be executed within the transaction based on LLVM.The basic principle is to insert an empty function before and after the function definition of the function to be executed in the transaction, and to call all the inserted empty functions in the program initialization phase. At this point, the code segment of the function to be executed in the transaction is loaded into memory along with the code segment of the empty function.We traverse all functions in IR PASS and add the corresponding before and after function definitions before and after the function definition.Then, tracking the function calls in all functions, in the corresponding before and after functions, adding the before and after function calls of the functions called in the original function.When there is no function call in the function, only

two empty functions are defined.Finally, we add calls to main_before and main_after during the initialization phase of the program to implement the call to all before and after functions during the initialization phase.

The above implementation still has some problems:

1.Due to the need to add a new function definition at the location of the function definition, the current solution cannot support automatic preloading of the library function. Our solution is to add a call to a library function such as memset when the program is initialized.For the same reason, the current solution cannot support the automatic preloading of LLVM functions such as llvm.var.annotation. Calling these functions will not result in page fault. We have solved this problem by adding a whitelist filter solution.

2.The current scenario cannot support automatic preloading of functions whose code segment size exceeds one page.According to the pre-loading mechanism, only the contents of the same page as the before and after functions are loaded into memory. If the size of the code segment of a single function exceeds one page, some contents may still not be loaded into memory. This situation will still result in pages fault within the transaction.

4.6 usage method

Users only need to add a small amount of code when using peapods to automatically implement sensitive information protection.When a user needs to add security enhancements to a certain cryptographic software system, (1) the __attribute__(annotation "Private_Key") keyword needs to be added before the definition of the sensitive variable of the type whose address has been determined during the compilation, (2) assign pointer-type sensitive variables to the args_list structure, then pass parameters to the START(args_list) and END(args_list) functions, (3) The START(args_list) function needs to be called before the variable assignment, and the END(args_list) function is called after the assignment.

In addition, the user needs to call the START(args_list) function before calculating the sensitive information and call the END(args_list) function after the sensitive information is calculated.When the sensitive information calculation is too time-consuming, the user needs to call the TEM(args list) function at the split location to split the transaction.

After users complete the above work, Peapods will automatically use HTM to protect sensitive information. Therefore, we can provide security enhancements for various types of cryptographic software systems without refactoring the entire cryptographic software in the presence of security vulnerabilities.The amount of code that the user needs to add is only a few hundred lines to several thousand lines (depending on the number and type of sensitive variables, the number of splits, etc.), the proportion compared to the total code volume of the cryptographic software system (usually several hundred thousand lines) is small.

It is worth noting that all user programs in the system need to be compiled with our modified LLVM compiler to ensure the security of the master key.

```
1  __attribute__((annotation"Private_Key")) unsighed char secret[16];
2
3  START(args_list);
4       secret={\
5            0x45,0x46,0x88,0x32,\
6            0x2a,0x6d,0x8c,0x31,\
7            0x58,0xf2,0x30,0x02,\
8            0x4f,0x32,0x7d,0x22\
9       };
10 END(args_list);
```

```
11
12 START(args_list);
13     secret_protect_compute_1();
14 TEM(args_list);
15     secret_protect_compute_2();
16
17     ...
18
19 TEM(args_list);
20     secret_protect_compute_n();
21 END(args_list);
```

5 SECURITY DISCUSSION

When the computer undergoes a core dump, as the Peapods transaction is interrupted, the master key, plaintext sensitive information, and intermediate calculations that have already been calculated are all cleared.Therefore, the attacker still cannot obtain sensitive information in plaintext.

Attackers may also attack sensitive information through the side channel.Fortunately, Peapods is immune to cache-based time-side channel attacks because AES-NI itself is not subject to any known side-channel attacks and the sensitive information calculation is fully implemented in the cache.

6 EVALUATION

We conducted an experiment to test the performance of Peapods.The experimental machine has an Intel Core i7-4770S quad-core processor running a Linux operating system (kernel version 3.13.1).We experimented with the PolarSSL cryptographic algorithm library and used Peapods to protect 2048-bit RSA private key calculations after splitting into 128 transactions and 256 transactions.The comparison objects in the experiment include: (1) the official default configuration of PolarSSL (version 1.3.9), (2) Peapods_128, the Peapods split into 128 transactions,(3) Peapods_256, the Peapods split into 256 transactions.

6.1 Local Performance

The local throughput for PolarSSL, Peapods_128, and Peapods_256 are: 391/s, 341/s, and 352/s (4 threads).Compared to PolarSSL, the performance of Peapods_128 and Peapods_256 dropped by 12.8% and 10.0%, respectively.The performance overhead introduced by Peapods mainly comes from: (1) waste of CPU clock cycles due to rollback of transactions; (2) encryption and decryption protection of sensitive information and intermediate calculation results; (3) preloading.The reason Peapods_256 performs slightly better than Peapods_128 is that the more granular splitting reduces transaction abort due to time-outs and other factors.

We also tested whether a memory-intensive program will have a greater impact on Peapods by running the Geekbench 3 memory stress test while Peapods is performing RSA private key calculations.In the experiment, 4 different memory stress tests will run on all CPU cores, which will cause about 10GB/s of memory data transfer.The maximum memory transfer rate supported by the machine when no user program is running is 13.7GB/s.Peapods_128 dropped from 341 RSA decryptions per second to 212 RSA decryptions per second, performance decreased by 37.8%, Peapods_256 decreased from 352 RSA decryptions per second to 218 RSA decryptions per second, performance decreased by 38.1%.PolarSSL decreased from 391

RSA decryptions per second to 241 RSA decryptions per second, performance decreased by 38.4%.

In short, the performance overhead of Peapods is acceptable. At the same time, Peapods perform better than other transaction-based protection due to preloaded optimizations.

6.2 Impact on Concurrent Processes

We used Geekbench 3 to test Peapods' impact on other concurrent processes.Figure X shows Geekbench 3 scores in multi-core mode and single-core mode.The baseline is a score in clean environment.

The Geekbench 3 benchmark program performed integer, floating point, and memory bandwidth tests, respectively.PolarSSL, Peapods_128, and Peapods_256 experiments have similar scores. Unprotected PolarSSL is slightly better than Peapods.When Geekbench occupies multiple cores, the load change that simultaneously processes RSA private key calculation requests cannot be ignoredthe baseline has a clear demarcation from other scores.It is worth noting that XMM registers are mainly used for floating-point calculations, and generally only 1-2 XMM registers are used.When the environment has only a small number of floating point calculations, we will not significantly affect the XMM7 register from the register allocation pool. When there are a large number of floating point calculations in the environment, the overall performance will be slightly reduced.In short, Peapods do not have a significant additional impact on other concurrent processes.

6.3 SGX

7 RELATED WORK

8 CONCLUSION

ACKNOWLEDGMENTS

The authors would like to thank XXX

REFERENCES

Received February 2007; revised March 2009; accepted June 2009