TRUSTWORTHY CROSS DSO CONTROL FLOW INTEGRITY

By

SOFIYA BAGODIYA

(Under the Direction of Mustakimur Rahman Khandaker)

ABSTRACT

Memory corruption is one of the oldest and most significant issues in computer security. To protect the vulnerabilities that arise from memory corruption, a mitigation technique called Control-flow Integrity (CFI) was developed. CFI has three main components: the CFI policy, reference monitor, and control flow graph (CFG). However, the most advanced context-sensitive CFI policies fall short of protecting real-world programs that require cross-DSO (Dynamic Shared Object) support. Our research proposes a placeholder CFG design integrated and enforced through a trusted enclave utilizing Intel Software Guard Extensions (SGX) and leveraged static value-flow analysis (SVF) to compute a partial CFG. This unique approach aims to enhance the protection provided by CFI, particularly for programs with cross-DSO dependencies, by employing a secure enclave and a customized CFG construction method.

INDEX WORDS: Control flow Integrity, Cross DSO CFI, context sensitive CFI, static value flow analysis, software guard extension.

TRUSTWORTHY CROSS DSO CONTROL FLOW INTEGRITY

by

SOFIYA BAGODIYA

Bachelors in computer science, Rajeev Gandhi Technical University, India, 2021

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2023

TRUSTWORTHY CROSS DSO CONTROL FLOW INTEGRITY

by

SOFIYA BAGODIYA

Major Professor: Mustakimur Rahman Khandaker

Committee: Le Guan
Suchendra M Bhandarkar

Electronic Version Approved:

Ron Walcott
Vice Provost for Graduate Education and Dean of the Graduate School
The University of Georgia
August 2023

DEDICATION

This thesis is dedicated to my loving parents as a token of my appreciation for their never-ending love, guidance, prayers, and sacrifices, which have shaped me into who I am today.

ACKNOWLEDGEMENT

I want to thank my major professor, Mustakimur Rahman Khandaker, for his unwavering support, invaluable guidance, and mentorship throughout this research journey. His expertise, encouragement, and dedication have been instrumental in shaping the outcome of this thesis.

I also sincerely thank the committee members for their valuable insights, constructive feedback, and collaborative spirit.

Furthermore, I wish to dedicate this accomplishment to my family, whose unconditional love and constant encouragement drive my achievements.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

The use of safe programming languages is growing, but Computer systems still run large amounts of legacy software written in unsafe languages, C and C++ because C/C++ offers strong performance and direct resource access; but one of the primary issues with C/C++ is the memory corruption vulnerabilities. vulnerabilities, such as buffer overflows, Code Reuse Attack etc. They increase the security risks to software systems. One common memory corruption vulnerability is a buffer overflow, which occurs when a program tries to write data beyond the allocated memory. Attackers use this vulnerability to overwrite important control flow pointers and redirect the program's execution to their malicious code (either by injecting new code or by reusing existing code sequences unintendedly), leading to control flow hijack[1, 2]

Various protection mechanisms were introduced again it, but the most famous one is the control flow integrity (CFI) mechanism[3-7] CFI ensures that the runtime control flow graph matches with the preconstructed control flow graph, which is created ahead of time. By ensuring that control transfers adhere to the predefined CFG, CFI mechanisms can effectively defend against control flow hijacking techniques, such as return-oriented programming (ROP) and jump-oriented programming (JOP). CFI creates the control flow graph using static analysis tools and at run time dynamically checks the run time flow with the preconstructed CFG. This technique can protect the indirect call and check whether the target is legitimate or not. If any changes in the CFG are detected, an alert is raised usually terminating the application, by this CFI can prevent from control flow hijacking

There are three important components of CFI mechanisms.

- **Control Flow Graph (CFG) Construction:** For a program, it represents all possible valid paths that a program can take.
- **CFI Policy:** This policy defines all the rules and regulations that control flow transfer should follow within the program. There are two types of policies: context-sensitives and context-insensitive policies.
- **Reference Monitor:** It implements the CFI policy. At runtime, it checks the control flow transfer against the preconstructed control flow graph to safeguard against any control flow hijacking attacks.

As CFI is giving promising results in protecting against control flow hijacking attacks, there are ongoing research efforts to improve its performance[8-10] , precision[9, 11-13] and applicability. Researchers have proposed various techniques, such as hardware-assisted CFI and context-sensitive CFI, to improve the preciseness and efficiency of CFI mechanisms. But there is a research gap present in CFI when it comes to the case of cross-Dynamic Shared Objects (DSOs). Cross DSOs case is complex because context, indirect calls, and targets are spread between the main program and the external libraries. Ensuring control flow integrity across DSOs requires techniques that can accurately validate the target for indirect calls even when the target is present in a different binary.

Table 1.1: Dependency Analysis of Benchmarks on External Libraries

Benchmarks		Dependency	
Name	**External Function**	**Name**	**Indirect call**
Grep	15	Musl	82
Bison	7	Musl	82
Openssh	7	Musl	82
Openssl	82	Musl	82

Table: 1.1 highlight popular benchmarks and their dependencies on external libraries for a significant number of functions. These functions are called by passing the function pointer as an argument, and from the library indirect call is made by using function pointer. as shown in table 1.1, Grep, Bison, and OpenSSL benchmarks depend on the Musl library, which involves 82 indirect function calls. Openssh, in addition to depending on Musl with 82 indirect calls, also requires OpenSSL. This dependency analysis table highlights the importance of validating targets for these indirect calls across shared libraries to ensure the security of an application.

Traditional Control Flow Integrity (CFI) mechanisms concentrate on those applications where a control transfer occurs within a single program and can be determined during the program's compilation. On the other hand, as the use of cross Dynamic Shared Objects (DSOs)[14] becomes more common, new challenges has raised as these DSOs are dynamic and interconnected. It creates a possible vulnerability because attacker can change the control flow by redirecting the indirect calls to a malicious code or an unintended function within other DSOs. The Standard CFI mechanism has limited visibility and control on control flow paths which is extended across multiple DSOs, thereby, making it compulsory to have a complete understanding

of the target. As a result, safeguarding applications against those control flow hijack attacks that have cross DSO boundaries, become more challenging.

This thesis introduces a solution to address the significant challenge of cross Dynamic Shared Object (DSO) control flow integrity. Our mechanism aims to safeguard applications from control flow hijacking, even in complex scenarios where the context, indirect call, and target spread across multiple DSOs. To achieve this, we adopt a comprehensive approach during the compilation process, by constructing a partial context-sensitive control flow graph (CFG) for the program and its associated libraries Using static value flow analysis tools, taking call sites as contextual information significantly improves the accuracy and precision of the Control Flow Graph (CFG). This partial CFG is then securely stored in read-only memory. During runtime, we establish a secure memory enclave utilizing software guard extensions. which dynamically constructs a complete control flow graph within this enclave by extracting and integrating the partial CFG obtained from the untrusted part outside the enclave. In order to provide security of reference monitor we used enclave memory.

This reference monitor continuously compares the runtime control flow graph of the program with the complete CFG stored within the enclave. If any attempt made by an attacker to modify the run time control flow reference monitor detect it and terminate the program.

We conducted an evaluation of our proposed mechanism. by creating partial control flow graph for the real-world projects, such as OpenSSL, OpenSSH, Musl, and Grep, and tested with our integrator algorithm which showcasing the potential of our approach in enhancing control flow integrity.

CHAPTER: 2

LITERATURE REVIEW

Control Flow Integrity (CFI) was firstly introduced[15] After that, numerous research has been done in the field[3, 7, 9, 16] However, this approach is imprecise due to the presence of many potential targets valid for each indirect call. It is a standard limitation of context-insensitive CFI systems, where the lack of contextual information hinders precise enforcement. One research that overcomes this issue is HyperSafe[12] HyperSafe is a context-insensitive CFI system that effectively addresses the problem by employing dedicated jump tables for each indirect call. Using these jump tables, HyperSafe significantly improves precision and reduces the number of potential targets for each indirect call. However, the primary purpose of HyperSafe is to enforce the CFI for a hypervisor. Accordingly, its performance overhead was not evaluated with the standard benchmarks, such as SPEC CPU2006.

While some CFI systems prioritize performance, this approach trades performance at the expense of security. For example, specific systems adopt an imprecise Control Flow Graph (CFG) approach[10, 13] which can leave them vulnerable to attacks. Even precise context-insensitive CFI systems can have vulnerabilities due to the large size of target sets[3] In a recent survey conducted[9] a comprehensive comparison of context-insensitive CFI systems was proposed. In contrast to these systems, the trustworthy cross-DSO CFI system implements context sensitivity CFI policy. Using call-site information as a context enhances security and improves the precision of Control Flow Integrity (CFI). Context-sensitive CFI achieves preciseness by incorporating contextual

information to differentiate sets of targets. However, incorporating context sensitivity into CFI systems presents challenges in system design, introducing trade-offs and opportunities. One such challenge is the need for context-sensitive Control Flow Graphs (CFGs). Context-sensitive points-to analysis, essential for context-sensitive CFGs, is known to have scalability issues. However, recent advancements, such as the release of SUPA (Scalable Path-Sensitive Points-to Analysis)[17] have significantly improved the scalability of points-to analysis. PathArmor[13] , PittyPat[11], and µCFI[11] implement the path-sensitivity CFI policy by taking the recent execution history recorded by Intel CPUs as contextual information. PathArmor utilizes the Last Branch Record (LBR) to capture the last sixteen branches taken by a process. On the other hand, PittyPat and µCFI utilize the Processor Trace (PT), which provides a more detailed record of the past execution.

LBR and PT are privileged and can only be accessed by the kernel. However, the transition into and out of the kernel is expensive. As a result, it is impractical to check these records for every Indirect Control Transfer (ICT). PathArmor selectively enforces the CFI policy at specific syscalls to overcome this limitation, protecting only a small part of the program. In contrast, PittyPat and µCFI redirect the trace to a separate process for control flow verification. While they can verify all ICTs, the enforcement of results is limited to the selected syscalls. This design has a drawback: the usable number of CPU cores is effectively reduced by half. Since all three systems cannot enforce the CFI policy at every ICT, our implementation of a trustworthy cross-DSO Control Flow Integrity (CFI) system incorporates a reference monitor for each ICT. While precise context-sensitive CFI systems have shown vulnerabilities in handling control transfers across Dynamic Shared Objects (DSOs), our solution aims to address these challenges.

One of the primary problems in dealing with cross-DSO control transfer is that ICT needs more visibility if an indirect call's target is present in a different binary. Hence, it creates a gap in the Control Flow Graph. To handle this issue, Clang has introduced a solution called cross-DSO mode. However, it has one significant issue. It is context insensitive.

In contrast, our proposed solution provides a more robust approach to providing trustworthy cross-DSO CFI systems with context-sensitive protection. These systems use call-site context to increase security by considering function call as a context.

With the use of call-site context for cross-DSO, our proposed solutions target to bridge the gaps in the CFG created by cross-DSO control transfer and increase the system's overall security. Our context-sensitive approach is more robust against control-flow hijacking attacks than the context-insensitive approach.

CHAPTER 3

MOTIVATION

3.1 Context-sensitive CFI

Traditional CFI solutions rely on context-insensitive policies, which statically derive a set of all valid targets for an indirect call without considering the program's execution history or context into consideration. While this approach is cost-effective, it has conceptual limitations as attackers can freely chain edges together, forming paths that may be invalid in the original control flow graph (CFG) and hijack the control flow graph. to overcome this limitation, context-sensitive CFI techniques have been introduced. In the context-sensitive approach, the past execution history is taken into consideration, such as the sequence of call sites that led to this ICT. This makes CFI more precise and protected. Call-site sensitive CFI is a CFI policy that provides a more precise control-flow enforcement approach by considering the call path as an context of control flow graph.

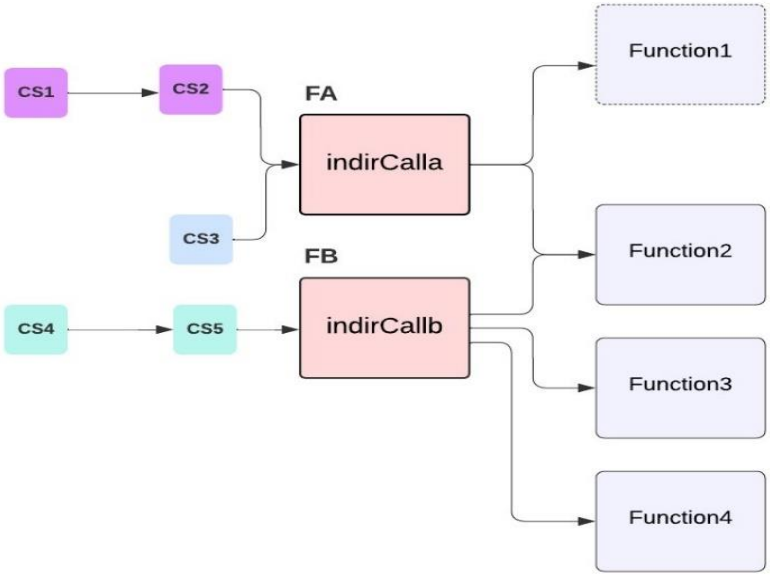

Figure 3.1: Control flow graph

As illustrated in figure: 3.1, there are two functions: first is Fa, which contains an indirect call called indirCalla. The second is Fb, which contains an indirect call called indirCallab. In a context-insensitive CFI policy, for indirCalla, without taking the past execution history into consideration, the valid targets would be {Function1, Function2}. Similarly, for indirCallb, valid targets would be {Function2, Function3, Function4}.

In contrast, a call-site sensitive CFI (CS-CFI) approach will consider one target as valid for each ICT by considering the call site as context. Hence, as shown in the figure 3.1, the control flow graph for the control flow integrity with a context-sensitive policy would be as follows.

Path 1: CS1 -> CS2 -> Fa -> IndirCalla -> Function1

Path 2: CS3 -> Fa -> IndirCalla -> Function2

Path 3: CS4 -> CS5 -> Fb -> IndirCallb -> Function3

Path 4: CS4 -> CS5 -> Fb -> IndirCallb -> Function4

These four paths illustrate valid execution paths for each Indirect Control Transfer (ICT) when using a context-sensitive policy.

Implementing context sensitive CFG can enhance the protection against control flow hijacking attacks by decreasing the size of valid target set for each indirect call. Considering additional information, known as context, at the time of the target verification process for each indirect call, a Context-Sensitive Control Flow Integrity (CS-CFI) gets the ability to identify and prevent control flow hijacking attempts that a context-insensitive approach might miss.

However, we have to take the drawbacks of using context-sensitive CFI into consideration too. One drawback is that it increases the complexity and runtime overhead by using context while creating CFG because tracking and collecting the execution context for each indirect call will require extra memory and computational resources. Additionally, this increase in complexity could pose implementation challenges for CS-CFI, making it more demanding when compared to context-insensitive CFI.

On the other hand, there are certain advantages of using a context-insensitive Control Flow Integrity (CFI) approach. Firstly, it is easier to implement and has a lower runtime overhead than context-sensitive CFI because it doesn't require tracking and collect the contexts. If the precise control flow analysis is not a requirement or when there are concerns about the additional overhead introduced by context tracking, context-insensitive CFI will be applicable as it provides protection against control-flow hijacking attacks without additional overhead.

3.2 Cross DSO Application

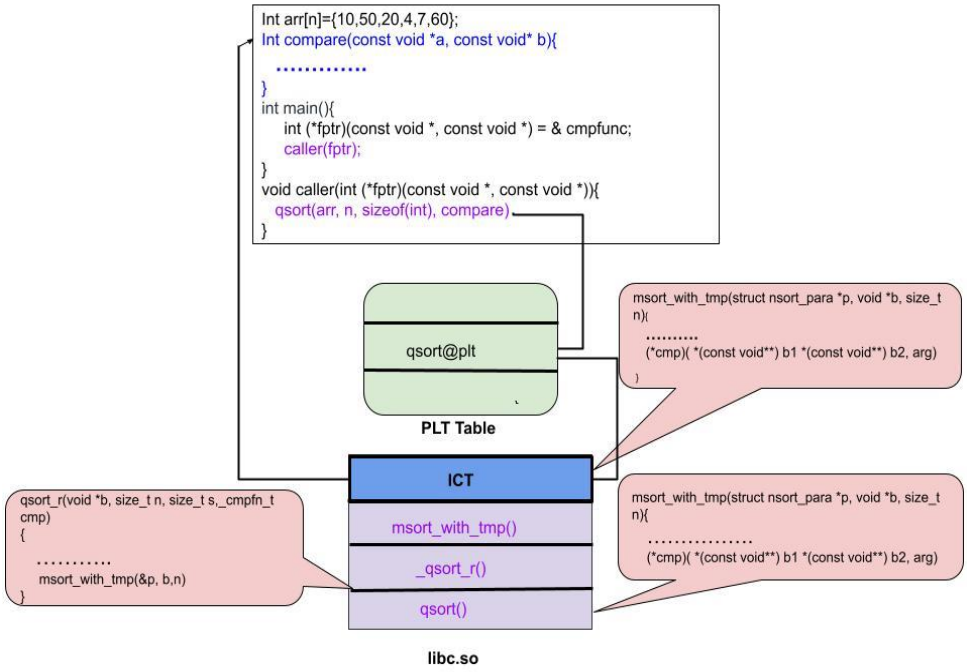

Fig 3.2: Demonstration of callsite-sensitive ICT using qsort()

Real-world applications often use external libraries to enhance their functionalities and capabilities. However, it becomes challenging to understand and organize the program's control flow when the context, indirect control transfer, and target are spread over different parts of the program. In such situations, the traditional method struggles to ensure the correctness of the program's control flow.

To gain more insight into this challenge, let's look at an example that involves the qsort() function. The main program uses the qsort function (a standard library to sort an array). To sort the array, qsort internally calls a special function called Comparator which compares the array elements and helps sort the elements correctly.

When the user's program calls the qsort function, a special table called Procedure Linkage Table (PLT) comes into the picture. It helps determine where the qsort function address. However, inside the Libc library, the qsort function doesn't directly do the sorting.

Instead, it calls another function called _qsort_r(), which then calls another function called msort_with_tmp() actually to perform the sorting process.

The user application's control graph includes two callsites: the place where the caller function is called and the place where the qsort function is called. In our case, the compared function is the target function. In contrast, the Libc library's qsort function has three callsites: qsort(), _qsort_r(), and msort_with_tmp(). Along with this, there is an indirect call to the comparator function from the Libc library. This demonstrates how the context is spread between two different parts of the program: the main program and the libs library.

There are several challenges in ensuring the control flow integrity across different shared objects. Firstly, the verification of the validity of call targets in DSOs for indirect calls because only the respective DSOs can confirm their correctness. Although, the target could be present in a different DSO, and the linking process could make the control flow graph even more difficult.

To overcome this challenge. The Clang compiler supports cross-DSO control flow integrity. However, it relies on context-sensitive policies. Although this approach provides some level of protection, it will not offer accurate and precise results in complex scenarios which involve multiple DSOs.

Despite the current advancements, ensuring control flow integrity in cross-DSO applications still remains a complex task, especially when context sensitivity is involved. One good solution is to compile all dependent libraries together with the main program. However, this approach is not efficient because it will increase the program's binary size.

3.3 Trustworthy enclave:

Intel SGX allows developers to create an extra set of CPU instructions to create isolated areas in computer hardware called enclaves, which protect sensitive information like passwords. From outside the enclave, we cannot read or write the enclave memory. Enclave protects the data even from the privileged operating system. Enclaved contents are cryptographically hashed by the CPU. It's not possible to debug the Production Enclaves with software or hardware debuggers. At every power cycle memory encryption key will be changed. The user defines the interface between the untrusted part and the trusted enclave in the EDL file. The EDL file specifies two types of functions:

- ECALLs (enclave calls, the functions that are called from an untrusted portion (outside application) to the trusted part(enclave))
- OCALLs (outside calls, the functions that are called from the trusted part(enclave) to the untrusted portion (outside application)

Enclave memory cannot access by the untrusted portion, so to communicate between the untrusted portion and enclave, SGX SDK generates a proxy function parsing the EDL file. The trusted functions included in the EnclaveProject_t.h and EnclaveProjct_t.c and EnclaveProject_u.h and EnclaveProject_u.c included untrusted proxy functions. The developer cannot access ECAll, and OCALl functions directly, user program will call the proxy function, and the proxy function will call the real Ecall function. When you pass the pointer from the untrusted part into the enclave, in the EDL file, you have to mention the direction of buffer references by the pointer is being passed into the call, out of the call, or in both directions with the size of the buffer. The proxy function checks that the range is not crossing the enclave boundary and substitutes the original

pointer with the new pointer referencing the buffer copy. This is the safe approach SGX used to marshal data and pointers between the untrusted portion and the enclave.

3.4 Static value flow analysis

Static value flow analysis is a handy tool for analyzing programs in a static manner. It is designed to be compatible with LLVM bitcode(.bc) files. These files provide an intermediate representation of programs that could be analyzed and optimized without needing a specific platform.

SVF utilized the LLVM infrastructure to carry out static program analysis. With this, it can create graph-based representations that capture different aspects of the program's behavior. This approach enables SVF to perform an in-depth analysis of programs, giving valuable information about how they operate.

For this thesis, we trace the value flow graph generated by the SVF. The VFG captures the flow of values and pointers within a program. It, providing insights into how values propagate through program statements and function calls by connecting the use of a function pointer to the definition of it. Interprocedurally value flow analysis is used by SVF to generate a value flow graph. We utilized SVF's Value Flow Graph (VFG) as a fundamental component of our analysis

CHAPTER 4

SYSTEM DESIGN

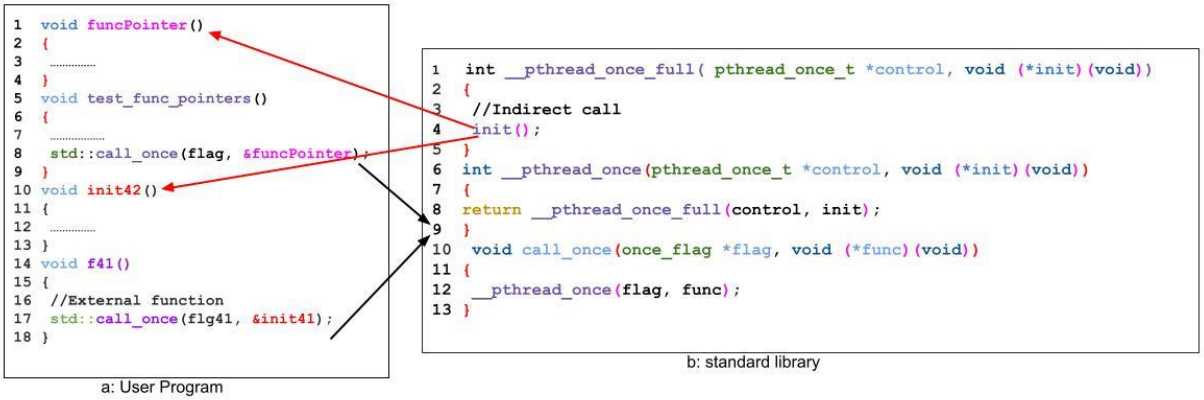

Fig 4.1: LLVM project calling standard library function call_once

4.1 Partial Control Flow Graph

Partial CFG captures the portion of complete CFG. Figure 4.1 shows the partial CFG for the case of the call_once() function. LLVM project calls the call_once(), which is present in the standard library. internal working of call_once() is it will call the _pthread_once_full() function, Within _pthread_once_full(), there is a call to a user-defined function. Calling funcPointer() and init42() from the standard library is an indirect call because the target is unknown at compile time; instead, the dereferencing of the function pointer is done at runtime.

For the standard library, the partial CFG included three call sites: std::call_once(), _pthread_once(), and _pthread_once_full() and an indirect call to the user-defined function. for the LLVM project, partial CFG included one call-sites call_once() and a target function funcPointer() and init42()

The complete CFG for the std::call_once() function is the addition of partial CFG from the LLVM project and partial CFG from the standard library. This case study shows how the complete CFG is spread between different DSO between the LLVM project and the musl library.

4.2 Trustworthy Integrator:

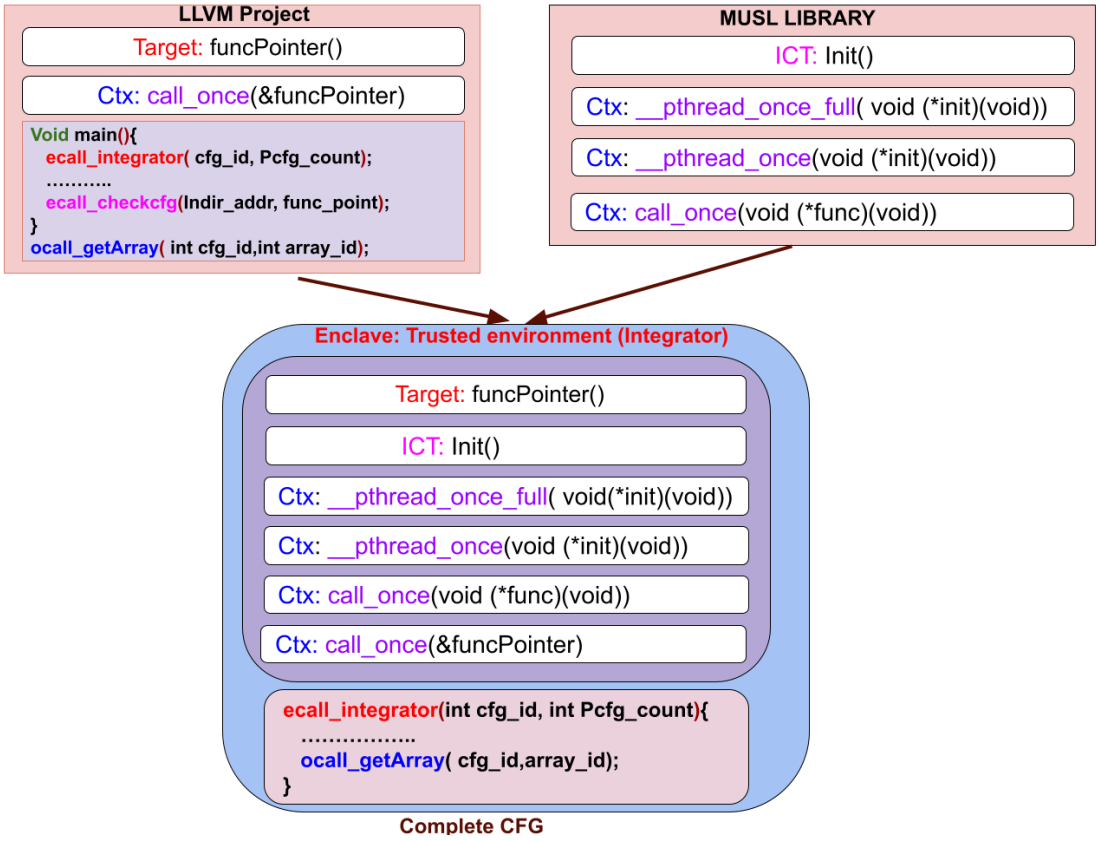

Fig 4.2: construction of trusted CFG from two partial CFGs

Figure 4.2 shows the construction of one complete CFG out of two partial CFGs. For the case study of the call_once function, we got two partial CFGs, one from the LLVM project where the call_once function is called and another PCFG from the musl library where the definition of call_once function is present. These partial CFGs only capture the portion of the complete control flow graph and thus need to be combined to construct a complete control flow graph. We created these complete CFGs inside the trusted memory

enclave, and for that, we instrumented a call to function init_sgx( ) as the first instruction. This function will create a safe memory enclave. A safe memory enclave is a secure hardware-based environment. The second instruction we instrumented inside the main function is a call to ecall_integrator(). It takes two arguments, the unique identifier for the CCFG and the number of PCFGs collected at compile time, which is two in this case.

The ecall_integrator() function is responsible for combining the PCFGs and constructing a CCFG, which captures the complete control flow of the program. The ecall_integrator() function makes a series of calls to the ocall_getArray() function to read each PCFG from the untrusted part, which is saved in read-only memory. OCall functions are used in Intel Software Guard Extensions (SGX) technology to securely communicate between the enclave and the untrusted part. Once the PCFGs are retrieved, the ecall_integrator() function combines them to construct a single CCFG, which captures the complete control flow graph and is stored inside the safe memory enclave's memory to maintain the integrity of constructed complete CFG.

4.3 Trustworthy Reference Monitor:

```
1  int __pthread_once_full(pthread_once_t *control, void (*init)(void))
2  {
3    ecall_checkcfg(int indirCall_Addr, init());
4    //Indirect call
5    init();
6  }
7  int __pthread_once(pthread_once_t *control, void (*init)(void))
8  {
9    return __pthread_once_full(control, init);
10 }
11 void call_once(once_flag *flag, void (*func)(void))
12 {
13   __pthread_once(flag, func);
14 }
```

Fig 4.3: Reference monitor

The reference monitor is one of the three important components of CFI. It matches the runtime flow of the program with the preconstructed control flow graph, which was created ahead of time. In this thesis, we have instrumented a reference monitor(ecall_checkcfg()) before each indirect call, as shown in Figure 4.3 ecall_checkcfg() function placed before the init() function(indirect call). ecall_checkcfg() function takes two arguments first, the address of indirect call instruction and the target function, run time call -site is collected by ecall_checkcfg() function using the __builtin_return_address() function, which retrieves the callers of the current function and combines it with the indirect call address and target, in this way, ecall_checkcfg() got runtime control flow graph and matched it against the complete control flow graph(CCFG) created by the integrator function by combining partial control flow graph. The CCFG represents the expected control flow path for the program. During execution, ecall_checkcfg() compares the runtime control flow graph with the expected CFG. If the run time control flow matches with the CCFG, the program continues its execution normally. However, if the control flow deviates from the expected path, indicating a potential compromise in control flow graph, the reference monitor terminates the program's execution. In this way, ecall_checkcfg() protects the indirect call against control flow hijacking.

CHAPTER 5

IMPLEMENTATION

5.1 Intel SGX Enclave

```
map<iCall, vector<dequeue<int>>> CFG_Map;
Ecall_integrator(CFG_id, nPCFG)
{
   dequeue<int> cCFG;
   for(i from 1 to nPCFG){
      pCFG = OCall_GetArray(i);
      cCFG.push_back(collect_call(pCFG));
      if(iCall= find_iCall(pCFG)){
         cCFG.push_back(iCall);
      }
      if(target= find_target(pCFG)){
         cCFG.push_back(target);
      }
   }
   fix_iCall_target(cCFG);
   CFG_Map[icall].push(cCFG);
}
hash_match(dequeue<int> oCFG, dequeue<int> rCFG>){
   return hash(oCFG)==hash(rCFG);
}
ecall_checkcfg(int indir_call, void *target_func)
{
   int callsites = __builtin_return_address(0);
   dequeue<int> rCFG;
   rCFG.push_back(callsites);
   rCFG.push_back(indir_call);
   rCFG.push_back(target_func);
   for (i from 1 to length(CFG_Map[indir_call]))
   {
      if (hash_match(CFG_Map[indir_call][i], rCFG))
      {
         return Success;
      }
      else
      {
         return Fail;
      }
   }
}
```

Fig 5.1: Trustworthy integrator and reference monitor

Figure 5.1 shows two important functions Ecall_integrator() and ecall_checkcfg(), which are placed inside the Trustworthy SGX enclave. These functions integrate the complete CFG and verify it against the run time flow. Keeping them inside the enclave enhances the integrity and security of the whole solution. Ecall_integrator() is responsible for integrating multiple partial control flow graphs (PCFGs) to construct a complete CFG.

It takes two arguments: CFGid, a unique identifier for each complete CFG, and the number of PCFGs to integrate. To retrieve the PCFGs from the untrusted part, the function OCall_GetPCFG() is called multiple times.The process of integration contains several steps. First, the collect_call() function is called by passing PCFG as an argument; this function collects all call sites from the PCFGs and adds them to the front of the complete CFG deque. Then, the find_iCall() function is called; this function identifies indirect calls within the PCFGs and appends it to the back of the deque. After that, the find_target() function is invoked, which identifies the target of the indirect call and pushes it to the back of the deque. Finally, the fix_iCall_target() function adjusts the structure of the CFG to ensure proper ordering firstly all the call-sites followed by indirect call and target.Since an indirect call can be associated with multiple complete CFGs, a map is created where the key represents the indirect call, and the value is a vector of complete CFGs. This allows for the efficient handling of multiple CFGs related to the same indirect call.

The second important function is the ecall_checkcfg(); it is responsible for checking the runtime control flow against the control flow graph constructed by Ecall_integrator() ahead of time. It takes two arguments: the indirect call and the target. To get the runtime CFG, the ecall_checkcfg() function retrieves the runtime call-sites using the __builtin_return_address() function and integrates it with the indirect call and target, and that is how run time CFG is found. Lastly, The ecall_checkcfg() function compares the runtime CFG with the previously integrated CFG to ensure that the control flow has not been manipulated by an attacker. By combining the capabilities of Ecall_integrator() and ecall_checkcfg() within the Trustworthy SGX enclave, the system can effectively integrate

partial CFGs into a complete CFG and subsequently verify the runtime CFG for security and integrity purposes.

5.2 Construction of pCFG:

In this thesis, we used a technique of static value flow analysis, Demand Driven Analysis (DDA) to optimize our analysis process. DDA focuses on specific parts of the program relevant to our queries, it helps in significant cost reduction and improved efficiency by creating a value flow graph from particular candidate, unlike the other approach of SVF in which whole program graph is evaluated. In our project, we define two candidates for DDA:

- **Candidate 1:** External function call with a function pointer argument. To identify this candidate, we iterate over the CallSiteSet, which contains a set of all the call sites in the program. At each call site, we check if the called function body is present within the binary. Using the isDeclaration() function in LLVM. Then we examine the function's arguments to identify function pointer types of argument. If we find a function pointer argument, we have our first candidate.
- **Candidate 2:** Indirect call with a public interface. Similar to Candidate 1, we iterate over the CallSiteSet and check if the call site is an indirect call using LLVM function isIndirectCall().

This candidate helps us to focus our analysis on the relevant parts of the program involving indirect calls with public interfaces and external function with function pointer as an argument. To construct the Partial Control Flow Graphs (PCFGs). As we traverse the value flow graph backward start from the given candidates,

We collected crucial information at various points and stored it within a vector for as a one partial CFG. During backtracking, specific nodes play significant roles. The LoadSVFGNode helps us capture call sites within the program, it represents memory accessing instructions that read values from address-taken objects. By examining the edges associated with each encountered node, we gain insights into control flow relationships and potential branching conditions.

When encountering nodes with multiple incoming edges, indicating the presence of multiple control flow paths, we split our vector in order to continue the analysis, ensuring that each vector represents a distinct path in the value flow graph.

This approach enables a more detailed analysis of the program's behavior. We traverse one edge and then return to the node where the edge splits to explore the other edge. Furthermore, the MSSAPHISVFGNode indicates the existence of multiple edges connected to this perticular node, helping in identify conditional statements within the control flow. Lastly, the AddrSVFGNode marks the completion of one partial CFG and indicates the particular node as an target for our analysis. In the case of indirect calls, a PCFG is considered complete even without a specific target, as our focus is on capturing the control flow paths and interactions within the program. On the other side, for external functions having function pointers as an argument, finding target for each valid PCFG is valuable. If a partial CFG does not have a target, we discard it during our analysis.

By using DDA technique and examining various nodes such as LoadSVFGNode, MSSAPHISVFGNode, and AddrSVFGNode, we construct PCFGs that effectively capture a particular potion of the program's control flow graph for the main program and all the dependent library.

CHAPTER 6

RESULT

Table 6.1: Benchmark Evaluation- calls, Dependencies, and PCFGs

Benchmarks	Externa Functions call	Unique External Function calls	PCFGs	Avg Call Sites	Dependency
openssl	82	22	81	3	musl
bison	7	4	8	1	musl
openssh	7	4	6	2	musl
grep	15	3	2	1	musl

Table 6.2: Indirect Call Analysis in the "musl" Library

Library	Indirect calls	Unique Indirect calls	PCFGs	Avg Call Sites
musl	82	18	78	3

In our evaluation, we analysis of four well-known benchmarks and the musl library. The results of our evaluation, presented in Table 6. 1, we observed that the benchmarks, on average, made a large number of external function calls, with an argument of function pointer type from one binary to another binary. We collected an average of 27 PCFGs for benchmark. The presence of multiple PCFGs shows the diverse control flow paths and interactions associated with these external function calls. We also evaluate the musl library on which all of these benchmarks have dependency. Within the musl library, we identified a large number of indirect calls, totaling 82. These indirect calls signify the

dynamic nature of function invocations within the library. Consequently, we collected 78 PCFGs associated with these indirect calls. Each PCFG has an average of three call sites, indicating the complexity and diverse control flow patterns present.When the complete CFG is spread between the project and the libraries, the complexity of creating a complete control flow graph increases.

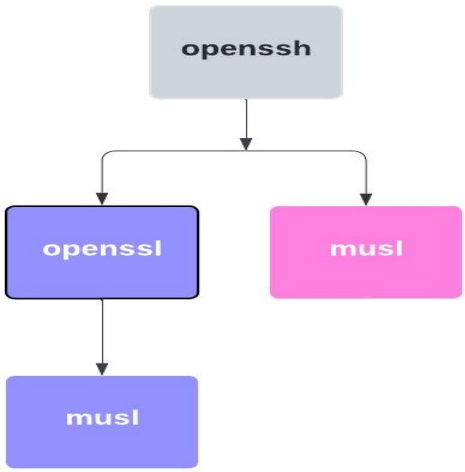

Figure 6.1: Dependency chart

Figure 6.1 shows the complexity of these benchmarks. The OpenSSH project depends on the OpenSSL library and the Musl library, while the OpenSSL library itself depends on the Musl library. Construction of the complete CFG is difficult because there is an interdependency between the project and the libraries.

Table 6.3: Integrator algorithm performance

Number of PCFGs	Average context	Time
3	3	0m0.115s
5	4	0m0.120s
7	5	0m0.130s

We checked our integrator algorithm by giving it to a varying number of partial control flow graphs with different numbers of contexts in it as shown in table 6.3. As the number of partial CFG increases, the algorithm seamlessly integrates them to construct complete CFGs, as shown in the table. The algorithm constructed CCFG by maintaining the order of all the call sites first, then indirect call and, the target efficiency.

CHAPTER 6

CONCLUSION

In this thesis, we have tackled the significant problem of maintaining control flow integrity across different shared objects by presenting a new method called a Partial Flow Graph. As the usage of unsafe programming languages such as C and C++ continues to increase, it is vital to address the security risks associated with memory corruption vulnerabilities. There are certain limitations in handling dynamic and interconnected control flow present in DSOs with the traditional control flow integrity mechanisms. Hence, there is a need for more robust and dependable solutions to enhance program security.

The motivation behind this work is to overcome the limitations of traditional CFI mechanisms, which primarily focus on monolithic binaries and struggle to effectively track and validate indirect function calls across shared libraries. This limitation creates a potential vulnerability, as attackers can exploit the dynamic and cross-boundary nature of DSOs to manipulate the control flow and execute malicious code. To fill this gape, this thesis proposes a solution by introducing a partial control flow graph. By using static value flow analysis during the compilation process, the system constructs partial control flow graphs (PCFGs) that capture the contextual information, indirect calls, and targets across the program and its associated libraries. This approach significantly improves the accuracy and precision of the control flow graph, using call-site as a context. The implementation of the proposed solution utilizes Intel SGX enclaves, which provide a secure memory enclave that safeguards sensitive data. Within the enclave, the complete control flow graph (CCFG) is dynamically constructed by integrating the partial CFGs

obtained from the untrusted part at compile time. This allows us to capture the entire control flow, including its interactions with various DSOs. The reference monitor, a crucial component within the enclave, continuously compares the runtime control flow graph with the CCFG to detect any unauthorized modifications and promptly halt the program's execution if control flow hijacking is detected. The evaluation of the proposed mechanism using real-world benchmarks, such as OpenSSL, OpenSSH, grep, bison, and the musl library, has demonstrated its effectiveness in detecting and mitigating control flow hijacking attacks. The results validate the successful integration of partial CFGs, the accurate tracking of control flow paths across DSO boundaries, and the prevention of unauthorized control flow modifications

CHAPTER 6

FUTURE WORK

We plan to enhance our research by creating a partial control flow graph using the origin-sensitive CFI policy in future work. Origin-sensitive CFI (OS-CFI) takes the origin of the code pointer called by an ICT as the context and constrains the targets of the ICT with this context. Also, we work on effective methods to extract information from the PLT table to adjust the partial CFGs. The PLT table contains crucial information about dynamically linked functions, and by incorporating this information into our analysis, we can get all the context of the control flow graph.

We enhance our analysis to handle the C++ program's virtual function calls because these calls can make things more complex as it involves late binding and dynamic dispatch. To handle this, we will develop a solution for the origin-sensitive CFI policy and call-site CFI policy to handle the virtual function calls more effectively.

To make sure our method is effective as well as efficient, we will conduct extensive testing on a wide range of real-world C++ programs that include industrial applications such as clang cross-DSO control flow integrity and open-source codes. Our primary focus will be on evaluating the key metrics such as accuracy and runtime overhead.

References

1. Sayeed, S., et al., *Control-flow integrity: attacks and protections.* Applied Sciences, 2019. **9**(20): p. 4229.
2. Szekeres, L., et al. *Sok: Eternal war in memory*. in *2013 IEEE Symposium on Security and Privacy*. 2013. IEEE.
3. Carlini, N., et al. *{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity*. in *24th USENIX Security Symposium (USENIX Security 15)*. 2015.
4. Niu, B. and G. Tan. *Per-input control-flow integrity*. in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015.
5. Criswell, J., N. Dautenhahn, and V. Adve. *KCoFI: Complete control-flow integrity for commodity operating system kernels*. in *2014 IEEE symposium on security and privacy*. 2014. IEEE.
6. Davi, L., P. Koeberl, and A.-R. Sadeghi. *Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation*. in *Proceedings of the 51st Annual Design Automation Conference*. 2014.
7. Davi, L., et al., *Building Control-flow Integrity Defenses.* Building Secure Defenses Against Code-Reuse Attacks, 2015: p. 27-54.
8. Zhang, C., et al. *Practical control flow integrity and randomization for binary executables*. in *2013 IEEE Symposium on Security and Privacy*. 2013. IEEE.
9. Burow, N., et al., *Control-flow integrity: Precision, security, and performance.* ACM Computing Surveys (CSUR), 2017. **50**(1): p. 1-33.
10. Tice, C., et al. *Enforcing {Forward-Edge}{Control-Flow} Integrity in {GCC} & {LLVM}*. in *23rd USENIX security symposium (USENIX security 14)*. 2014.

11. Ding, R., et al. *Efficient Protection of Path-Sensitive Control Security*. in *USENIX Security Symposium*. 2017.

12. Wang, Z. and X. Jiang. *Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity*. in *2010 IEEE symposium on security and privacy*. 2010. IEEE.

13. Van der Veen, V., et al. *Practical context-sensitive CFI*. in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 2015.

14. *Clang 17.0.0git documentation CONTROL FLOW INTEGRITY*. Available from:https://clang.llvm.org/docs/ControlFlowIntegrity.html.

15. Abadi, M., et al., *Control-flow integrity principles, implementations, and applications.* ACM Transactions on Information and System Security (TISSEC), 2009. **13**(1): p. 1-40.

16. Khandaker, M., et al. *Adaptive call-site sensitive control flow integrity*. in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2019. IEEE.

17. Sui, Y. and J. Xue, *Value-flow-based demand-driven pointer analysis for C and C++.* IEEE Transactions on Software Engineering, 2018. **46**(8): p. 812-835.