The Pennsylvania State University
The Graduate School

AUTOMATIC EDL GENERATION FOR INTEL SOFTWARE GUARD EXTENSIONS

A Thesis in
Computer Science and Engineering
by
Eralp Sahin

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

August 2020

Abstract

Intel Software Guard Extensions (SGX) is a hardware-assisted Trusted Execution Environment for desktop and server platforms. SGX increases the security of an application by isolating a specific part of an application code and memory. This isolated part is called an *enclave*. Enclaves are protected from other processors running at higher privilege levels and the operating system. The interface between the untrusted zone and the enclave in the application is defined in the Enclave Definition Language (EDL) file. Converting a conventional application to an SGX application is a non-trivial process. Programmers need to define the interface manually in the EDL file. Additionally, SGX creates proxy routines for cross-domain functions in the application that have different signatures than the original functions so there needs to be a refactoring of the source code. We developed a tool to automate and reduce the work needed for defining the interface and refactoring. Our tool performs a static analysis on the original application to obtain information about the user-defined types and the pointer parameters of the cross-domain functions in the application. It generates the EDL file for the application and refactors the original source code to comply with the proxy routines generated by SGX.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my gratitude to my advisor Gang Tan who was always encouraging throughout my studies at the Pennsylvania State University. I would like to thank the members of my committee for their time and patience. Special thanks to Yongzhe Huang without whom this work would never be achievable.

Dedication

Lovingly dedicated to my beautiful wife Elif. Thank you for always listening to me when I talk about my thesis, trying to help and supporting me.

Chapter 1

Introduction

Processing some kind of private information is a common part of modern software applications. These secret data should be protected from unauthorized access. To achieve this, security sensitive applications often apply data encryption themselves on top of operating system's own security enforcements such as limiting access to applications and files of other users. These mechanisms fell short of defending the private information on a computer where malicious parties have administrative privileges since it is possible to gain unauthorized access to an application's memory where decrypted secret data live.

Trusted Execution Environment (TEE) aim to solve this security problem by offering an isolated environment. This isolation can be done on both hardware and software level. Intel Software Guard Extensions (SGX) is a hardware-assisted TEE for desktop and server platforms. SGX allows creation of an *enclave* (trusted container) and offers a protection from a set of known hardware and software attacks [1]. Some of these attacks are mentioned in the section 2.1.

An SGX application consists of two parts called *trusted* and *untrusted* components. The communication between these zones (interface) is defined in an Enclave Definition Language (EDL) file.

Converting a conventional application to an SGX application is a non-trivial process. To automate and reduce the programmers work we developed a tool for automatic EDL file and SGX code generation. The attributes for parameters of cross boundary functions (ECALLs and OCALLs) in EDL are gathered from a static code analysis consisting of *parameter access analysis* from *IDL-GEN* [2] and some other heuristics. The automation of code refactoring uses information gathered from the code analysis to refactor and inject necessary code snippet to the application. The rest of this thesis is organized as follows. Next sections describe in detail the problem, the motivation, and the system of our tool. Chapter 2 summarizes information about the background and related work of the tool. Chapter 3 presents the methodology used in the static analysis of the tool. Chapter 4 illustrates results on experimental benchmarks. Chapter 5 discusses limitations of the heuristics in the static analysis and future work for improvements.

1.1 Problem

A conventional application needs major code refactoring to comply with the requirements of the SGX SDK. In addition to the EDL for the interface, every ECALL and OCALL in the source code needs to be refactored with additional parameters for the SGX proxy routines. Moreover, SGX does not support all GCC built-ins including commonly used functions such as `printf` and `strcpy`. In such cases, the programmer needs to manually refactor the code. Making these additions and changes in the application code is manual and laborious even though the information and the necessary work to be done for refactoring can be deduced by a static analysis.

1.2 Tool workflow

The tool takes an intermediate representation (IR) in the form of LLVM IR for the trusted and the untrusted components of the application as an input. After compiling the program through `clang` resulting IR files can be linked together with `llvm-link` in accordance with the separation of the trusted and the untrusted boundaries. For the analysis we first construct a program dependence graph (PDG) of the whole application. Analysis of the tool including *parameter access analysis* and heuristics uses the PDG for gathering the access information, definitions of user-defined types, and other information to be used during the EDL generation for the application. Basics of the enclave development and the EDL are described in section 2.1 and 2.1.2 respectively. In addition to the EDL file, the tool generates wrapper functions for each ECALL, OCALL, and unsupported functions to comply with the SGX SDK. Section 2.1.3 has a table consisting of most common functions in the unsupported functions list. The tool refactors each call sitte for ECALLs and OCALLs and injects necessary code to the application code such as enclave initialization. Diagram of the tool is shown in figure 1.1

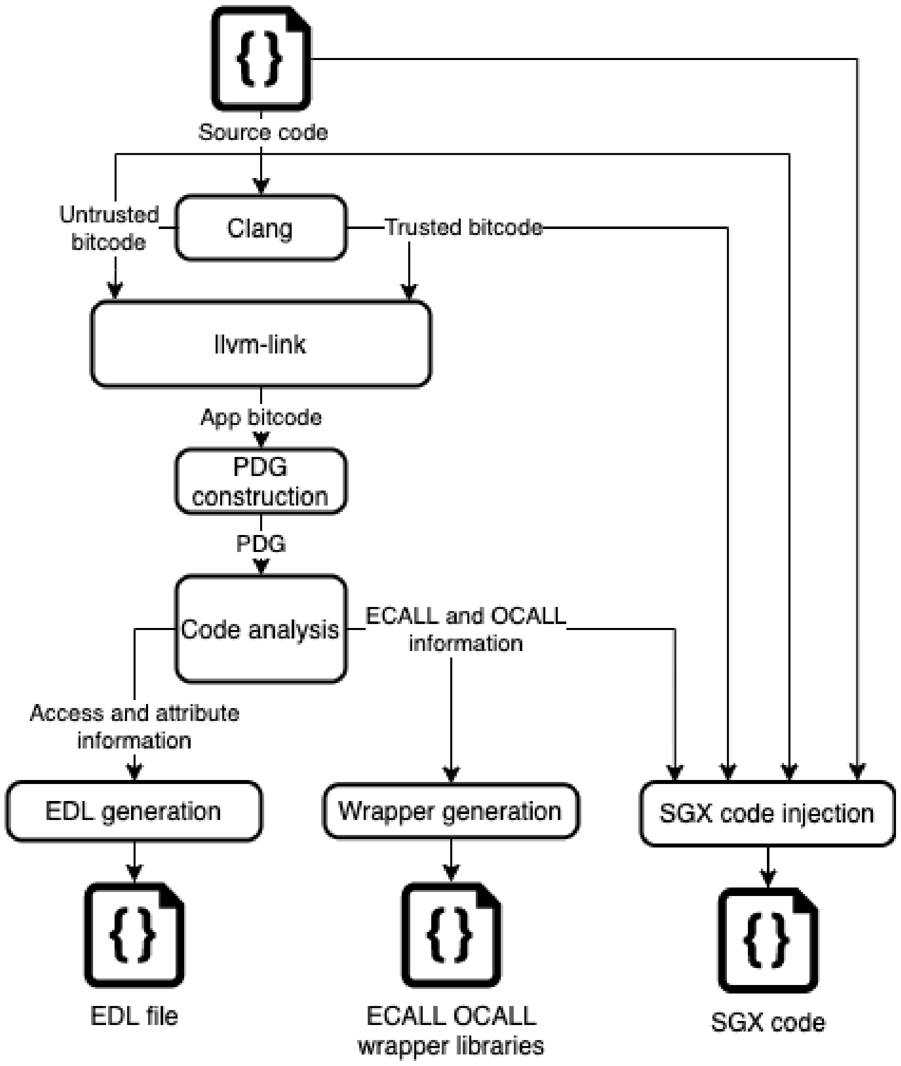

Figure 1.1. Workflow

1.3 Motivation

Following example demonstrates the workflow of the tool. This Example is a toy C application whose functionality is to get a plaintext from the user and encrypt the text string according to a key generated using a pseudo random number generator `rand`. For the sake of simplicity we can assume the trusted and untrusted boundaries are separated in such a way that the secret key initialization and the encryption are supposed to be executed in the enclave. Figure 1.2 shows the untrusted component of the application and the figure 1.3 shows the enclave of the application.

```
int main() {
  char text[1024];

  printf("Enter plaintext: ");
  scanf("%1023s", text);

  initkey(strlen(text));
  char* ciphertext = encrypt(text, strlen(text));
  printf("Cipher text: ");
  for (int i = 0; i < strlen(text); i++)
    printf("%x ", ciphertext[i]);
  return 0;
}
```

Figure 1.2. Example untrusted code

To convert this application into an SGX application, we need to have an EDL file for the interface between the components. SGX SDK includes a tool called *Edger8r* that generates proxy routines for both the untrusted components and enclaves using the EDL files. Details of *Edger8r* are described in section 2.1.1. The generated proxy routines for the ECALLs and the OCALLs have different signatures than the original functions so there needs to be a refactoring of the code. We generate the EDL files and a library of functions for the purpose of refactoring with our tool. Then the SGX code injection module of the tool injects definitions and initialization code necessary for the SGX SDK and refactors each ECALL and OCALL site by changing the called functions name to the automatically generated

```
char *key;
void initkey(int sz) {
  key = (char *)(malloc(sz));
  for (int i = 0; i < sz; i++) key[i] = rand() % 5;
}

char *encrypt(char *plaintext, int sz) {
  char *ciphertext = (char *)(malloc(sz));
  for (int i = 0; i < sz; i++)
    ciphertext[i] = plaintext[i] ^ key[i];
  return ciphertext;
}
```

Figure 1.3. Example trusted code

library of functions.

In our example, the cross boundary functions are `encrypt` and `initkey` which are called from the untrusted component of the application. The tool computes these *imported functions* for the untrusted component using the same methodology used in *IDL-GEN*. While we do not have any imported functions in enclave in our example, the tool would collect if there were any. EDL file will be consisted of the interface specifications for the functions in these two distinct imported functions lists.

Parameter access analysis is performed on these function lists again similar to *IDL-GEN*. A summary of what the analysis results is described in section 3.1. In addition to the access analysis we analyze the code for static information about buffer size, user-defined types, and other specific usages of pointers in code.

The EDL file generated for our example is shown in figure 1.4. In the EDL code, each cross boundary function has an interface specification. For pointer type parameters the tool generates attributes including the access information and the size according to the requirements of the SGX SDK. `Unsupported.edl` is an SGX library we use for the unsupported built-in functions, sections 2.1.2 and 3.3 discuss the generated EDL code in more detail. In addition to the EDL file, the tool generates libraries for untrusted components and the enclave composed of wrapper functions for the cross boundary functions as shown in figure 1.5 and 1.6.

As mentioned before, this step is necessary because in an SGX application, the

```
enclave {

  trusted {
    public char* encrypt([in, count=1024] char* plaintext,
                         int sz );
    public void initkey( int sz );
  };

  from "Unsupported.edl" import *;
  untrusted {

  };
};
```

Figure 1.4. Example EDL

```
#include "Enclave_u.h"
#include "sgx_urts.h"
#include "sgx_utils.h"
extern sgx_enclave_id_t global_eid;
char* encrypt_ECALL( char* plaintext, int sz);
void initkey_ECALL( int sz);
```

Figure 1.5. Example ECALL library header

```
#include "Ecalls.h"
sgx_enclave_id_t global_eid = 0;
char* encrypt_ECALL( char* plaintext, int sz) {
  char* res;
  encrypt(global_eid, &res, plaintext, sz);
  return res;
}

void initkey_ECALL( int sz) {
  initkey(global_eid, sz);
}
```

Figure 1.6. Example ECALL library implementation

untrusted and trusted components call proxy routines of the functions instead of directly calling the functions and proxy routines have a different signature.

Apart from the EDL file and the libraries for ECALL and OCALL wrapper functions, the ECALL and OCALL sites in the source code in both untrusted and trusted zones need to be refactored so that the function calls will call the wrappers. The result of the refactoring of the untrusted component is shown in figure 1.7. Since there are no OCALLs, trusted code is not changed other than adding necessary headers on top.

```
#include "Ecalls.h"
#include "Unsupported.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  if (initialize_enclave(&global_eid,
                         "enclave.token",
                         "enclave.signed.so") < 0) {
      printf("Failed to initialize enclave.\n");
      return 1;
  }

  char text[1024];

  printf("Enter plaintext: ");
  scanf("%1023s", text);

  initkey_ECALL(strlen(text));
  char* ciphertext = encrypt_ECALL(text, strlen(text));
  printf("Cipher text: ");
  for (int i = 0; i < strlen(text); i++)
    printf("%x ", ciphertext[i]);
  return 0;
}
```

Figure 1.7. Example code refactoring

The changes that are made by the tool are (1) necessary headers for the wrapper library and the unsupported function library; (2) enclave initialization code at the first line in the scope of the function `main`; (3) function call changes in the source code so that the wrapper functions will be called instead of the original functions.

Chapter 2

Background and Related Work

2.1 Intel Software Guard Extensions

Intel SGX enables protection of secrets in the application. SGX is a set of CPU instructions in the Intel architecture that operates by allocating hardware protected memory in which both the code and the data can reside. An SGX enabled application consists of two main components called *trusted (enclave)* and *untrusted*. The data within the enclave can only be accessed by the code that resides in the enclave and the code can be invoked from the untrusted components with special instructions that SGX provides in Intel architecture. This procedure provides security properties such as:

- Enclave memory cannot be accessed from outside the enclave regardless of privelege level and CPU mode.
- Production enclaves cannot be debugged by any kind of debuggers.
- As mentioned before, enclave code cannot be invoked without special instructions, only way is to call an enclave function *ECALL* via SGX instruction. This renders register manipulation and stack manipulation attacks ineffective.
- Enclave memory is protected by encryption. The encryption key is randomly generated and stored in the CPU with no access.

SGX requiress two logical components in an application, *trusted* component resides in the enclave and can access the protected memory. This component is also called *enclave*. *Untrusted* component is the rest of the application including the operating system.

While it is possible to call an enclave function from the untrusted zone, the enclave can also invoke a function from the untrusted zone. These functions in each zone that can be called cross boundary are called *ECALL* for functions in enclave and *OCALL* for functions in untrusted zone. The interface of these calls should be described in an EDL file in the application.

An enclave needs to be created in the SGX application. `sgx_create_enclave` function is provided in SGX to load the enclave using its filename, its configuration and the user-defined enclave id [3].

2.1.1 Edger8r

Edger8r is a tool in the SGX toolset that generates secure proxy routines to ensure data marshalling for ECALLs and OCALLs according to the interface specification in the EDL file [3].

Given an EDL file `demo.edl` Edger8r generates the following files:

- `demo_t.h`: Prototypes of ECALLs.
- `demo_t.c`: Definitions of ECALLs.
- `demo_u.h`: Prototypes of OCALLs.
- `demo_u.c`: Definitions of OCALLs.

The generated functions have different signatures than the original ones. Figures 2.1 and 2.2 show an example for the original function from the enclave and the untrusted proxy generated for the function by Edger8r.

```
int bar() {
  return 42;
}
```

Figure 2.1. Example ECALL

```
sgx_status_t bar(sgx_enclave_id_t eid,
                               int* retval) {
  sgx_status_t status;
  ms_bar_t ms;
  status = sgx_ecall(eid, 0, &ocall_table_Enclave, &ms);
  if(status == SGX_SUCCESS && retval)
    *retval = ms.ms_retval;
  return status;
}
```

Figure 2.2. Example untrusted proxy generated by Edger8r

In the SGX application the original ECALL function is not accessible outside the enclave. Untrusted components only have access to the untrusted proxy of the ECALL. Edger8r changes the signature of the function. First parameter `sgx_enclave_id_t eid` is a unique ID for the enclave which is used as a handle to an enclave by these proxy functions. Next, the second parameter is used as a replacement for the return type. In our example, the second parameter is `int *` since the function is supposed to return an integer. Edger8r then adds the original functions parameters in the original order if there are any. The enclave id parameter only exists for ECALLs. Figure 2.3 shows the call and return diagram for when the untrusted zone calls a function from the enclave called `bar`.

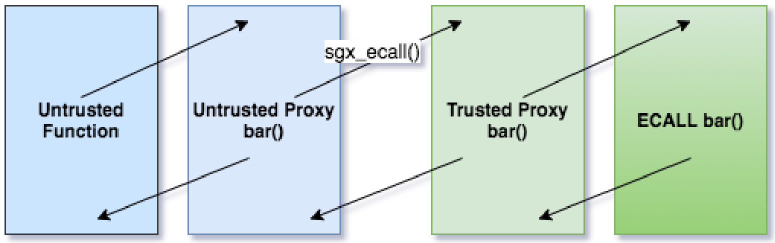

Figure 2.3. ECALL call and return diagram

The `sgx_ecall` function call is the bridge between the untrusted zone and the enclave. First argument is the `eid` of the enclave. Second parameter is the index of the ECALL function in the table SGX uses internally. According to the order in the EDL file of the application, Edger8r assigns indices to functions. In our example, `bar` function is the first in the EDL, as a result its index is 0. Third parameter is a table that stores OCALLs. SGX sends this table to the enclave side to be used when enclave makes an OCALL call. Finally, the last parameter is a struct used for data marshalling by the SGX. As shown before, the return value is handled by this struct if there are any.

2.1.2 Enclave Definition Language

Enclave definition language describes the interface between the untrusted zone and the enclave. As mentioned before, Edger8r uses the information from the EDL files to create proxy routines. EDL syntax is an extension to standard C/C++ syntax with additional information about the types, parameters, and the visibility for the ECALLs and OCALLs. Figure 2.4 shows the syntax and features of EDL.

```
enclave {
  include "mytypes.h" // Include user-defined types
  struct myStruct { // Add user-defined type definitions
    int data;
    const char* text;
  };
  trusted { // Optional if file is imported in another edl
    public void some_func([in] struct myStruct* st);
    void duplicate_string([in, string] const char*  str1,
                          [in, out, string] const char* str2);

  };
  from "Library.edl" import *;
  untrusted { // Optional
    // Allow non-public ECALLs to be called
    void ocall_printf([in] struct myStruct* st)
         allow(duplicate_string);
  };
};
```

Figure 2.4. EDL syntax

Edger8r requires the definitions of user-defined types for generating proxy routines. EDL can include headers for the user-defined type definitions (structs, unions, enums, typedefs). Additionally, structs, enums, and unions can also be defined directly in the EDL file. An EDL file can also be used as a library and be imported from another EDL file to add its functions to an enclave. Figure 2.4 shows the import syntax.

The first ECALL is annotated with the keyword `public` to enable that ECALL to be directly called from the untrusted zone. The second ECALL is private and can only be called from an OCALL by granting permissions explicitly with the keyword `allow`. ECALLs define entry points into the enclave while OCALLs define the transfer of control from the enclave to untrusted zone. An enclave in an SGX application should have at least one public ECALL.

2.1.2.1 Attributes

Pointer parameters are annotated with attributes in EDL. Specifically for direction and size information. Following are the attributes defined in EDL:

- `[in]`: This attribute is used for when the parameter is passed from the caller to the called function. SGX handles marshalling by creating a pointer in the called side and copying the buffer pointed by the pointer.

- `[out]`: This attribute is used for when the parameter is returned from the called function to the caller.

- `[user_check]`: In contrast to the direction attributes above, `user_check` can be used to trade performance for protection. Using this attribute does not provide any kind of data marshalling. Additionally, SGX verifies that the buffer from untrusted zone is pointing to an address in the untrusted zone and vice versa when used with direction attributes. With `user_check` attribute the raw pointer address will be passed and the programmer should do the bounds checking on the address if needed.

- `[size]` and `[count]`: The proxy routines copy the buffer pointed by the pointer and in order to copy the buffer contents the routine needs to know how much data needs to be copied. The total number of bytes are calculated by `count * size`. If `size` is not specified it is assumed to be `sizeof(pointer type)` for pointer types other than `void`. For this reason it is often used only for `void` pointers. For `void` pointers this attribute is required. If `count` is not specified it is assumed to be equal to `1`. These attributes can have a literal value, such as `[in , count=10]` and can be used with a parameter, such as

```
void foo([in, size=len] void* ptr, size_t len);
```

- `[string]`: This attribute indicates that a parameter is a `NULL` terminated `char` pointer C string. Proxy routine determines the length of the string implicitly. `[in]` attribute is required for this attribute.
- `[isptr]`: If a pointer type is aliased to a type that does not have an asterisk(*) EDL does not recognize the pointer and the parser generates an error. This attribute explicitly indicates that it is in fact a pointer type.
- `[readonly]`: Similar to `isptr` attribute a user-defined type of a pointer to a const data type should be annotated explicitly with this attribute for EDL.
- `[isary]`: This attributed is used to indicate that the user-defined parameter is an array.

2.1.3 Unsupported Built-in Functions

SGX has a special trusted C library that does not contain any function that are considered insecure [3]. Table 2.1 shows some of the most common functions that are not supported in enclave. These functions can still be called from the untrusted zone of the application.

strcat	strcpy	strdup	stpcpy
fprintf	printf	scanf	fscanf
vprintf	vsprintf	rand	exit

Table 2.1. Unsupported C functions

2.2 Program Dependence Graph

Our analysis depends on a program dependence graph (PDG) of the application. A PDG is a single graph consisting of data dependence graph (DDG) and control dependence graph (CDG). Since PDG connects the graphs of computationally related parts of the program, many analyses including parameter access analysis and analysis for our heuristics require less time to perform than with other representations [4].

Nodes in the graph are *instruction nodes* that represent instructions in the program. Edges represent data/control dependence and call edges. Our main focus for our analysis is the data dependence edges. The *parameter tree* approach in our PDG construction for representing pointer data that are passed during function calls obviates the need for global pointer analysis [5].

2.2.1 Parameter tree

We build a *formal parameter tree* for each parameter of a function. The parameter tree contains nodes that represent memory regions that can be accessed through the parameter. Additionally, *actual parameter tree* is constructed for each argument at a function call site and connect nodes in an actual tree with corresponding formal tree nodes.

```
struct st {int a, float b};
void foo (struct st* s);
```

Figure 2.5. Example struct definition and prototype

Figures 2.5 and 2.6 shows the parameter tree generated for a sample code in which a struct pointer to a struct with two fields is passed to function `foo`.

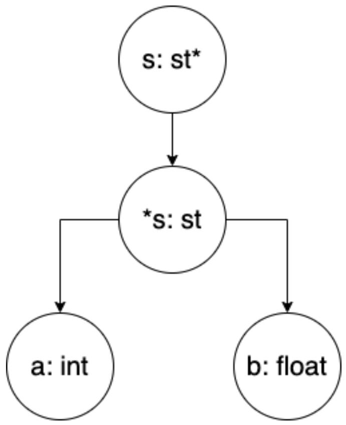

Figure 2.6. Example Parameter tree for parameter s

In the parameter tree each node represents a memory region. The final tree includes node for the pointer to the struct, node for the struct, and one node for each fields of the struct. Parameter tree simplifies the computation of inter-procedural data dependence. Intra-procedural analysis for each function can be composed together using parameter trees to build a PDG for a large program. Details can be found in the Ptrsplit [5].

Chapter 3

Code Analysis and Generation

The application takes the LLVM IR of the application which is created by linking the two partitions and performs a static analysis to generate the outputs. First step of the tool is to construct a PDG of the application. Our tool computes the cross-domain function sets for both the untrusted zone and the enclave. Afterwards, parameter access analysis and other heuristics are performed for the functions in these sets on the PDG. Final step is to generate EDL file and the wrapper libraries for the application based on the findings in the static analysis.

3.1 Parameter access analysis

We use parameter access analysis on pointer type parameters of cross-domain function sets. It calculates how and which memory regions in the parameter tree are used [2]. Using a worklist based approach, for each function in the cross-domain function set and its call graph we perform an intraprocedural parameter access analysis and perform an interprocedural parameter access analysis for the call graph of the function until the analysis reaches a fixpoint. Access analysis computes a set of *access labels* (*READ* — *WRITE*) for the parameters of a function. Details can be found in the *IDL-GEN* [2].

3.2 Heuristics

Algorithm 1 represents the overal intraprocedural analysis of our tool. We will describe the methods we use in the following sections.

Algorithm 1 Intraprocedural access and attribute analysis

```
    Input: $arg$ is an argument, $T$ is the parameter tree
    Output: Access and Attribute Map $AM$
 1: $AM \leftarrow \emptyset$
 2: procedure GetIntrAArgInfo($arg, T$)
 3:     for node $n$ in $T$ do
 4:         $AM[n] \leftarrow \emptyset$
 5:         $IS \leftarrow \{i \mid$ instructions that access n's memory region$\}$
 6:         for Instruction $inst$ in $IS$ do
 7:             if $inst$ is StoreInst and $n$ is destination then
 8:                 $AM[n] \leftarrow AM[n] \cup WRITE$
 9:             else if $inst$ is LoadInst and $n$ is source then
10:                 $AM[n] \leftarrow AM[n] \cup READ$
11:             else if $inst$ is CallInst and $n$ is operand then
12:                 if $arg$ is $char*$ then
13:                     analyzeString($AM, n, arg, inst$)
14:                 else
15:                     analyzeSize($AM, n, arg, inst$)
16:                 end if
17:             end if
18:         end for
19:     end for
20:     analyzeCount($AM, arg, G$)
21: end procedure
```

3.2.1 [string]

EDL syntax has a special attribute for `NULL` terminated `char` pointer C strings. During the deep copy of the contents of the pointer the size of the data is calculated by the proxy routine instead of relying on `[size]` and `[count]` attributes. We inspect the call instructions during the parameter access analysis for parameters of `char` pointer type. We compare the called function name against a combination of a pre populated set of function names and the functions in the application that our tool already analyzed. Table 3.1 shows our pre populated string function set. If a parameter of a function is detected to be a `NULL` terminated `char` pointer C string, similar to access analysis this information is used in the interprocedural parameter access analysis.

Algorithm 2 String attribute

Input: *arg* is an argument, *n* is a parameter tree node, *callInst* is the instruction in which *arg* used as an operand

Output: Access and Attribute Map *AM*

```
1:  procedure ANALYZESTRING($AM, n, arg, callInst$)
2:      $func \leftarrow$ callee in $inst$
3:      $num \leftarrow$ operand number in $inst$
4:      if $func$ in $ReadStrSet$ and $num$ is $ReadStrSet[func]$ then
5:          $AM[n] \leftarrow AM[n] \cup STRING$
6:      end if
7:      if $func$ in $WriteStrSet$ and $num$ is $WriteStrSet[func]$ then
8:          $AM[n] \leftarrow AM[n] \cup STRING \cup WRITE$
9:      end if
10:     if $func$ in $G$ and corresponding $param$ in $func$ is $STRING$ then
11:         $AM[n] \leftarrow AM[n] \cup STRING$
12:     end if
13:     if $func$ in $printfFormatSet$ and $num$ corresponds to $\%s$ then
14:         $AM[n] \leftarrow AM[n] \cup STRING$
15:     end if
16:     if $arg$ is $STRING$ and corresponding $param$ in $func$ is $char*$ then
17:         $n' \leftarrow$ parameter tree node of $param$
18:         $AM[n'] \leftarrow AM[n'] \cup STRING$
19:     end if
20: end procedure
```

[string] [in]		[string] [out]	
strlen	strnlen	-	-
strcmp	strncmp	sprintf	snprintf
strcoll	strtok	-	-
strchr	strrchr	-	-
strpbrk	strspn	-	-
strcspn	strstr	-	-
strdup	strndup	-	-
strcat	strncat	strcat	strncat
strcpy	strncpy	strcpy	strncpy
open	fopen	-	-
strftime	vsnprintf	strftime	vsnprintf

Table 3.1. `[string]` attribute functions set

According to the SGX specifications, pointer parameters with `[string]` attribute are required to also have the `[in]` direction attribute. Our algorithm adds the label `STRING` which implicitly means `[in, string]` attributes. Our algorithm also adds `WRITE` label to arguments that are used with functions in `WriteStrSet`. In this case, `[in, out, string]` attributes would be generated for the argument in the EDL file.

Our tool presumes the parameter is a `NULL` terminated `char` pointer C string if it is used as an argument to a function from the table above. According to the index of the argument in the call instruction we infer whether the string is read or written. For example, our tool sets `[out]` attribute to the first argument in a `strcat` call and `[in]` attribute to the second argument. For `printf` style function calls, our tool parses format strings if they are literal values. We can also detect that a `char` pointer is a string if we know that it is used with the `%s` specifier. Algorithm 2 presents the cases where we detect `[string]` attribute. The `ReadStrSet` is the set of functions on the left side of the table 3.1 and `WriteStrSet` is the right side.

Figure 3.1 demonstrates an example for the algorithm. We have two functions `foo` and `bar` with two `char` pointer parameters. We can assume `foo` is the root ECALL for simplicity.

When our tool analyzes `foo`, it infers that `buf1` is a string by finding the `strlen` in the `ReadStrSet`. Since our tool obtained that `buf1` is a string, corresponding

```c
void foo(char* buf1, char* buf2) {
    size_t size = strlen(buf1);
    bar(buf1, buf2);
}

void bar(char* buf3, char* buf4) {
    strcat(buf4, "!");
}
```

Figure 3.1. Example code snippet for `[string]` attribute

parameters in all the functions in `foo`'s transitive closure that gets the `buf1` as an argument are also inferred to be a string. This is handled on line 16 in the algorithm. The `buf3` parameter is an example to this in our case.

For the parameter `buf2`, only usage is a `bar` call and our tool have not analyzed that function yet. When intraprocedural analysis analyzes `bar`, it infers that `buf4` is a string since the function is in the `WriteStrSet` and `buf4` is the first parameter. Handled on the line 7 in the algorithm, we add the label `WRITE` to the map of the argument. Our tool annotates this parameter with `[in, out, string]` attribute. At this point, when `buf2` is analyzed again, the tool can infer that it is a string because `bar` is in our constructed PDG and the corresponding parameter (`buf4`) is a string.

```c
void foo(const char* str1) {
    ...
    const char* str2 = str1;
    strlen(str2);
    ...
}
```

Figure 3.2. Indirect usage of the argument with a string function

We are analyzing direct usages of parameters in the function and do not track the local variables. While it is possible to track all the variables related to the parameter such as assignment to new local variables, this increases the complexity of the PDG analysis because the parameter is only indirectly connected to the values of the variables it is assigned to in our constructed PDG. Figure 3.2 a code snippet and 3.3 the LLVM IR of the code show an example for this case where our tool cannot infer `[string]` correctly.

```
...
%4 = load i8*, i8** %2, align 8, !dbg !18
store i8* %4, i8** %3, align 8, !dbg !17
%5 = load i8*, i8** %3, align 8, !dbg !19
%6 = call i64 @strlen(i8* %5), !dbg !20
...
```

Figure 3.3. Indirect usage of the argument with a string function LLVM IR

Our PDG has an edge from `%4` to store instruction and we should analyze the store instruction to infer `%5` is in fact related to `%4`. This process increases the complexity of the PDG analysis.

3.2.2 [size]

Our tool uses a similar approach for the `[size]` attribute. As we mentioned before, this attribute is often used only for `void` pointer types since it is required. `[count]` attribute can still be used together with `[size]` for `void` pointers. We use a set of functions specific to `void` pointers that we can detect the size from. For example our tool recognizes the size for the `ptr` parameter in the ECALL shown in figure 3.4 from the `memset` call.

```
void clear_mem(void * ptr, size_t len) {
    memset(ptr, 0, len);
}
```

Figure 3.4. Usage of memset with size len

It is trivial to extract the size information from the third argument of the `memset` call if it is a literal value. For the other case where the size is sent as a parameter we make use of the PDG to analyze all parameters of the function to deduce that the second parameter of `clear_mem` is the size of the pointer. Our tool assigns the `[out, size=len]` attribute to the `ptr` parameter for the figure 3.4. Algorithm 3 presents this process. Table 3.2 shows our pre populated function set. The `ReadVoidSet` is the set of functions on the left side of the table 3.2 and `WriteVoidSet` is the right side.

[in]		[out]	
memcpy	memmove	memcpy	memmove
memchr	memrchr	memset	-
memcmp	write	-	read

Table 3.2. [sıze] attribute functions set

Similar to [string] attribute we infer whether the argument is read or written according to the index. For example first and second argument of memcpy call should be annotated with [out] and [in] attributes respectively.

For a pointer parameter that is used in a call instruction, our tool first checks the read and write pre populated sets to find the called function. If it finds in any of the sets the index of the parameter in the instruction and the value from the pre populated set is compared. If they also match, our tool assigns either [in] or [our] attribute to the node of the parameter.

Additionally, we analyze the size operand of the function. For example, third argument of a memset call is used to obtain how much data in the buffer is filled. We presume that this is the total size of the buffer. If the third argument is a literal value, our tool uses literal value with the [size] attribute. Otherwise, it tries to relate any of the parameters of the function with the size argument of the call instruction. We analyze the PDG for the read dependencies of the parameters and if any is related to the size argument of the function call, our tool annotates the [size] attribute with that parameters name. This method falls short when a local variable is used with the arguments value indirectly similar to the weakness we mentioned in the [string] section. Figure 3.5 shows the LLVM IR of the clear_mem function from the figure 3.4.

Algorithm 3 Size attribute

Input: *arg* is an argument, *n* is a parameter tree node, *callInst* is the instruction in which *arg* used as an operand

Output: Access and Attribute Map *AM*

```
 1: procedure analyzeSize($AM, n, arg, callInst$)
 2:     $func \leftarrow$ callee in $inst$
 3:     $num \leftarrow$ operand number in $inst$
 4:     if $func$ in $ReadVoidSet$ and $num$ is $ReadVoidSet[func]$ then
 5:         $AM[n] \leftarrow AM[n] \cup READ$
 6:     end if
 7:     if $func$ in $WriteVoidSet$ and $num$ is $WriteVoidSet[func]$ then
 8:         $AM[n] \leftarrow AM[n] \cup WRITE$
 9:     end if
10:     if $func$ in $WriteVoidSet$ or $ReadVoidSet$ then
11:         $sizeOp \leftarrow$ size operand of $callInst$
12:         if $sizeOp$ is literal then
13:             $AM[n] \leftarrow AM[n] \cup [size =$value of $sizeOp]$
14:         else
15:             $caller \leftarrow$function of $arg$
16:             for parameter $param$ in $caller$ function's parameter list do
17:                 $DS \leftarrow \{\ dep \mid READ$ dependency list of $param$ in $PDG\ \}$
18:                 for dependency $dep$ in $DS$ do
19:                     if $dep$ is $sizeOp$ then
20:                         $AM[n] \leftarrow AM[n] \cup [size = param]$
21:                     end if
22:                 end for
23:             end for
24:         end if
25:     end if
26: end procedure
```

Analyzing all parameters in the `clear_mem` function, our tool finds that the dependency list of `%4` which corresponds to `len` parameter includes `%6` which is the third argument of `memset` call. In this case, we annotate the size attribute with the parameter name.

```
define void @clear_mem(i8*, i64) #0 !dbg !49 {
    ...
    %4 = alloca i64, align 8
    ...
    %6 = load i64, i64* %4, align 8, !dbg !63
    call void @memset(i8* align 1 %5, i8 0, i64 %6)
    ret void, !dbg !65
}
```

Figure 3.5. LLVM IR of `clear_mem` function

3.2.3 [count]

Finally the `[count]` attribute is also used for determining how much data needs to be deep copied in the proxy routines. While `[size]` is determined by the SGX automatically for pointer types other than `void` pointers, if `[count]` attribute is not specified it is assumed to be 1. Our tool analyzes call instructions to `malloc` in the application and occurences of `getelementptr` (GEP) instructions to extract the buffer size information. The syntax of a sample GEP instruction from LLVM IR is shown below.

```
<result> = getelementptr inbounds <ty>, <ty>* <ptrval>, [inrange]
                                <ty> <idx>*

%16 = getelementptr inbounds [1024 x i8], [1024 x i8]* %3, i64 0,
                              i64 0, !dbg !53
```

For statically defined arrays, the `[count]` can be obtained from the GEP instruction alone, using the type information in the instruction. In this case, the attribute value will be a literal and we don't need to analyze the PDG further for the parameters of the function. Our tool can obtain `[count]` information if the `malloc` argument is a literal or it is a variable and is sent to the cross-domain function with the pointer.

Other cases such as variable argument in `malloc` not sent to the cross-domain function and `malloc` call inside the cross-domain function instead of its caller are not analyzed and we cannot infer the `[count]` in this case. Algorithm 4 presents the method our tool uses for static arrays and `malloc` function calls.

Figure 3.6 is an example to demonstrate how the algorithm works for different cases with the `[count]` attribute.

```
void foo(int* a) {
    ...
}
void bar(int* b, int count) {
    ...
}
int main() {
    int* ptr = (int*)malloc(1024);
    int buffer[1024];
    int n = 512;
    int* ptr2 = (int*)malloc(n);

    foo(ptr); // Pointer
    foo(buffer); // Static array

    bar(ptr2, n);
}
```

Figure 3.6. Example code snippet for `[count]` attribute

The algorithm analyzes all the callers of the cross-domain function for obtaining `[count]` attribute of the *arg* parameter. Focusing on `foo` and `int* a` the algorithm will inspect the `main` function. Inside the `main` it inspects each function call instruction in the `main` function. If the call instruction is calling `foo` and the argument corresponding to `int* a` is a result of a GEP instruction, we analyze the type information inside the GEP instruction to infer that it is a static array and the size of the array.

Algorithm 4 Count attribute

```
    Input: $arg$ is a parameter, $func$ is the function, $G$ is a PDG
    Output: Access and Attribute Map $AM$
 1: procedure ANALYZECOUNT($AM, arg, G$)
 2:     $CS \leftarrow \{$ $caller$ | caller functions of $func$ in $G$ $\}$
 3:     for Function $caller$ in $CS$ do
 4:         $callSet \leftarrow \{$ $callInst$ | call instructions in $caller$ $\}$
 5:         for Call instruction $callInst$ in $callSet$ do
 6:             $callee \leftarrow$ called function in $callInst$
 7:             if $callee$ is $func$ then
 8:                 $op \leftarrow$ operand corresponding to $arg$ in $callInst$
 9:                 if $op$ is $GEP$ then
10:                     $AM[arg] \leftarrow AM[arg] \cup [count =$value of $op]$
11:                 end if
12:             else
13:                 for Malloc call $mallocCall$ in $callSet$ do
14:                     $res \leftarrow$ result of the $mallocCall$
15:                     if $arg$ is dependent of $res$ then
16:                         $mallocArg \leftarrow$ argument in $mallocCall$
17:                         if $mallocArg$ is literal then
18:                             $AM[arg] \leftarrow AM[arg] \cup [count =$value of $mallocArg]$
19:                             $AM[arg] \leftarrow AM[arg] \cup [size =1]$
20:                         else if $mallocArg$ is an operand in $callInst$ then
21:                             $param \leftarrow$ name of the parameter corresponding to
    $mallocArg$
22:                             $AM[arg] \leftarrow AM[arg] \cup [count =param]$
23:                             $AM[arg] \leftarrow AM[arg] \cup [size =1]$
24:                         end if
25:                     end if
26:                 end for
27:             end if
28:         end for
29:     end for
30: end procedure
```

During the intraprocedural analysis of the `int* b` parameter in `bar` function, our tool will not be able to find the count from a GEP instruction. It will analyze all the `malloc` calls in the callers of `bar` which is only `main`. Checking the dependencies of the results of `malloc` calls, if `int* b` is a dependent of a result, we use the argument in the `malloc` call to infer the count. For the function `bar` it is not a literal, it is a variable that is also used as an argument in the `bar` function call. In this case, our tool annotates the `[count]` attribute with that parameters name.

For total amount of bytes calculation SGX multiplies `[count]` with `[size]` and uses `sizeof(pointer type)` for unspecified sizes, the total amount of bytes would be incorrect if we only annotate `[count]` for the cases we use `malloc` calls. `malloc(n)` allocates only n many bytes, for this reason we explicitly assign `[size]` attribute to 1 to result in the correct amount of bytes.

3.3 EDL Generation

EDL files must have at least one public *(root)* ECALL if they are not a library EDL. The tool computes the transitive closure of the `main` function that does not expand into enclave functions. When a call to an enclave function is seen, our tool adds the function to the closure but does not expand into that function. This results in a transitive closure that consists of only untrusted functions and the root ECALLs.

A private ECALL can only be called from an OCALL if the OCALL has explicit permission granted with the `allow` syntax in the EDL. The tool checks the OCALLs transitive closure similar to the procedure above to determine which ECALLs should be allowed explicitly. If an ECALL is in the OCALLs untrusted transitive closure and the ECALL is not a public ECALL, permission to invoke the ECALL is granted to the OCALL.

Edger8r requires the user-defined type definitions in the EDL file. EDL supports definitions of structs, enums, and unions but does not support `typedef` syntax. Using `DICompositeType` metadata in LLVM IR our tool generates definitions of the composite types in the EDL including nested composite types. For `typedef` types we make use of the include statements supported in EDL.

3.4 SGX Code Generation

Since the Edger8r tool creates a proxy routine with a different signature for a function, the application needs refactoring. Our tool generates and injects this necessary code into the application. Figure 3.7 shows the signature of a simple ECALL function and the signature of the proxy routine created by Edger8r for the function.

```
// Signature of the function in original application
char *encrypt(char *plaintext, int sz);

// Signature of the proxy routine for untrusted domain
sgx_status_t encrypt(sgx_enclave_id_t eid,
                     char** retval,
                     char* plaintext,
                     int sz);
```

Figure 3.7. Function and proxy routine signatures

In the figure above, the return type is changed and two additional parameters are added to the function. Our tool creates a wrapper function by adding an `_ECALL` or `_OCALL` suffix to the function name. Figure 3.8 shows the wrapper function generated by our tool.

```
    char* encrypt_ECALL( char* plaintext, int sz) {
        char* res;
        encrypt(global_eid, &res, plaintext, sz);
        return res;
    }
```

Figure 3.8. ECALL wrapper

For ECALLs in the application our tool generates a library of wrapper functions in files `Ecalls.h` and `Ecalls.cpp`. Similarly for OCALLs our tool generates `Ocalls.h` and `Ocalls.cpp`. Our tool inspects the source code and changes the code by injecting necessary suffix to ECALLs and OCALLs to use these wrappers instead of the original functions. This automates the conversion from a regular function call to an SGX ECALL or OCALL.

We are using our Sample SGX project[1] configured to simulation mode together with this tool. Sample project is already structured for the outputs of this tool and includes a library of OCALLs for built-in functions that are unsupported in SGX. For example, while `printf` is unsupported in the enclave side, it is implemented as an OCALL in our library to eliminate unnecessary refactoring of the original application.

Finally as mentioned before, an SGX application first needs to create an enclave using functions provided by the SGX SDK. This creation can be done at the beginning of the `main` function. During the SGX code generation phase, our tool injects a code that includes a call to a custom function that handles enclave creation in our sample project.

[1]`https://github.com/eralpsahin/sample-sgx`

Chapter 4

Experiments

Our tool requires a separation boundary in the application. For experimental purposes we assume each function is a cross-domain function and generate the interface specifications for each function in the application. We evaluated the tool on the `thttpd`, `htpasswd`, and `mini_httpd`.

Table 4.1 presents the size of each benchmark.

Application	lines of code	function count	parameter count	pointer parameters
thttpd	7041	144	266	172
mini_httpd	3331	72	99	67
htpasswd	171	8	15	10

Table 4.1. Size of the benchmarks

4.1 `[string]` inference

Table 4.2 shows the number of `char` pointers and number of string attributes our tool finds.

Application	char* parameters	[string]	[user_check]
thttpd	59	39	6
mini_httpd	44	33	4
htpasswd	7	2	0

Table 4.2. String attribute results

As shown by the data, our inference of `[string]` attribute is possibly not complete. We will discuss the results for each application.

4.1.1 `mini_httpd`

Our heuristics and parameter access analysis do not find any information for 4 `char` pointers resulting in `[user_check]` attribute for those parameters.

Upon inspecting the `mini_httpd` source manually, we see that `[user_check]` cases can be inferred as `[string]` if we do a full PDG analysis. Also, we gather the following information for the remaining 7 `char` pointer parameters that our tool inferred direction attributes but did not infer `[string]`;

- Out of 7, 1 buffers size is inferred with `[size]` attribute and it is not a C string.
- Out of remaining 6, 4 of the char buffers' contents are either directly or indirectly checked against null termination in a flow such as a loop. Our tool was able to infer `[count]` attribute for 2 of those 4. We are including these in the false negative set since even though we inferred the `[count]`, they are strings. Additionally, remaining 2 parameters could be inferred with a full PDG analysis because arguments to the functions were also used with string functions but since we are not analyzing local variables of functions and uses of null termination our tool did not infer `[string]`. Figure 4.1 demonstrates the case where full PDG analysis would improve our inference.

```
void foo(char* ptr1, char* ptr2) {
    ...
}
int main() {
    char l[256];
    char w[256];
    strcpy(l,w);
    foo(l, w);
}
```

Figure 4.1. Local variables used in string function

In total, there are 10 `char` pointer parameters that are in fact C strings but our tool did not infer. Our false-negative rate is 10 out of 44.

4.1.2 `htpasswd`

The `htpasswd` is a small application with only 7 `char` pointers. Our tool infers `[string]` for 2 of the `char` pointers. After manually investigating, it is clear that out of the remaining 5, 4 of them are also used as C strings and a full PDG analysis could make the correct inference, similar to the example in figure 4.1. Our tool infers `[count]` for those 5 `char` pointers from the static array size analysis.

In total, there are 4 `char` pointer parameters out of 7 that are in fact C strings but our tool did not infer. Our false-negative rate is 4 out of 7.

4.1.3 `thttpd`

We first analyze the `char` pointer parameters with `[user_check]` attribute. We see that all 6 parameters are C strings. Full PDG analysis including local variables would improve our inference results. Out of 59 `char` pointers, our tool infers `[string]` for 39 and use `[user_check]` for 6, we will analyze the remaining 14 parameters.

- Out of 14, 12 `char` pointer parameters are C strings and could be inferred with a full PDG analysis.
- Our tool was able to infer `[count]` for the remaining 2 from static array information. They are in fact C strings and should be inferred as such.

In total, there are 20 `char` pointer parameters out of 59 that are in fact C strings but our tool did not infer. Our false-negative rate is 20 out of 59.

Table 4.3 shows the quality of our `[string]` inference.

False negative here is the number of `char` pointer parameters that are in fact C strings that our tool misses. False-negative rate is the number of false negative cases over all `char` pointer parameters. False positive is the number of `char` pointer parameters that our tool misclassifies as strings. In our experiments we did not encounter a false positive.

Application	**False Negative**	**False Positive**
thttpd	34%	0
mini_httpd	23%	0
htpasswd	57%	0

Table 4.3. Quality of `[string]` inference

4.2 `[size]` and `[count]` inference

Excluding the `char` pointers that are inferred as or were supposed to be inferred as `[string]`, pointer parameters used with direction attribute should have `[size]` or `[count]` attributes. Ideally every pointer parameter should have direction attributes for data marshalling, but there are cases where we cannot find any access information. Our tool uses `[user_check]` for the parameters that it cannot find access information. For some cases `[user_check]` attribute is used correctly such as when passing a function pointer from the untrusted zone to an OCALL through an ECALL. The raw address is needed in this case and ECALL will have no access information for the pointer other than sending it to an OCALL. Since there will be no access information, our tool annotates the parameter with `[user_check]` in this case. The results of the buffer size inference we make on those parameters with `[count]` and `[size]` are shown in table 4.4.

Application	**non string pointers**	**[count]**	**[size]**	**[user_check]**
thttpd	113	6	4	17
mini_httpd	24	0	4	7
htpasswd	4	1	0	3

Table 4.4. Size and count attribute results

4.2.1 mini_httpd

Out of 67 pointer parameters 43 of them are `NULL` terminated C strings and there are 24 pointer parameters that either require size attributes along with direction attributes or a `[user_check]` attribute. Our tool was able to infer `[size]` for 4 parameters and was not able to infer access information for 7 parameters.

The remaining 13 are pointer parameters that we inferred direction attribute but could not infer `[size]` or `[count]` attributes.

`[user_check]` attribute is used for pointers with no access information. Since we do not compute PDG for library function calls and instead represent them as regular instruction nodes, we don't infer access information for such cases. As a result we cannot infer size attributes. Figure 4.2 demonstrates an example. `modf` function modifies the second parameter. `int* intpart` parameter should have `[out]` attribute. But since we represent library calls as instruction nodes instead of constructing PDG of them, our tool annotates the parameter only with `[user_check]` attribute.

```
void modfWrapper(double* intpart) {
  ...
  double fractpart = modf(value, intpart);
  ...
}
```

Figure 4.2. Library call represented as instruction node

We gather the following information by our inspection;

- Out of 7 `[user_check]` attributes 2 are union pointers and no size attribute is needed. Remaining 5 need size attributes if they were to used with a direction attribute.

- Out of 13 pointer parameters with either `[in]` or `[out]` attribute 11 of them are pointer to variables or structs and not buffers which SGX assumes to have count of 1. Our inspection shows that the count is in fact 1.

- Last 2 of the pointers' sizes are handled with `realloc` in their transitive closure which we do not analyze for currently.

In total there are 7 pointer parameters that are in fact buffers and need size attributes if they are used with a direction attribute but our tool did not infer. Our false-negative rate is 7 out of 24.

4.2.2 `htpasswd`

There are only 4 non string pointer parameters and 3 of them are `FILE` pointers with `[user_check]` attribute and not buffers. Our tool infers the `[count]` of the remaining pointer parameter from static array size analysis. Our false-negative rate is 0 out of 7.

4.2.3 `thttpd`

There are 113 non string pointers, our tool infers `[count]` for 6 and `[size]` for 4 of these parameters. 17 of them are annotated with `[user_check]` because our tool was not able to infer access information.

- 9 pointer parameters out of 17 `[user_check]` are pointers to variables or structs and not buffers.
- The address in 2 pointer parameters out of 17 `[user_check]` are used directly in the function. Figure 4.3 demonstrates an example for this usage. If used with direction attribute, `addr` would be a deep copy and would not store the same address.
- Remaining 6 out of 17 `[user_check]` need size attributes if they were to used with a direction attribute.

```
void foo(void * addr) {
  ...
  void * ptr = (void*)realloc(addr, 10);
  ...
}
```

Figure 4.3. Direct usage of pointer address

There are 86 pointer parameters that our tool inferred direction attribute but could not infer `[size]` or `[count]` attributes.

- Out of 86 pointer parameters with either `[in]` or `[out]` attribute 84 of them are pointer to variables or structs and not buffers which SGX assumes to have count of 1. Our inspection shows that the count is in fact 1.
- Remaining 2 pointer parameters are buffers that are used with our pre populated `[size]` function set. Our tool was unable to infer the size because instead of using directly, some kind of arithmetic calculation was used and our tool does not analyze this case. Such as

```
read( ..., (void*) buf + nread, nbytes - nread );
```

In total, our tool was unable to infer size attributes for 8 pointer parameters out of 113 that need these attributes. Our false-negative rate is 8 out of 113.

Table 4.5 shows the quality of our size attributes inference.

For the `[count]` attribute our tool inferred instead of `[string]`, we have 2 false positives out of 67 pointer parameters we analyzed in `mini_httpd`, 5 false positives out of 7 pointer parameters we analyzed in `htpasswd`, and 2 false positives out of 172 pointer parameters we analyzed in `thttpd`.

Application	**False Negative**	**False Positive**
thttpd	07%	1%
mini_httpd	29%	3%
htpasswd	0	71%

Table 4.5. Quality of `[size]` and `[count]` inference

Chapter 5

Discussion and Future Work

In this thesis we have presented a tool for for Intel SGX that can generate EDL files and automate the code refactoring needed to convert regular C application to an SGX application.

We have implemented heuristics using static analysis for `[string]`, `[count]`, and `[size]` attributes. We demonstrated the workflow of the tool from the inputs to the outputs. While the performance is not the main concern, since we often analyze the whole applications PDG for the heuristics, the performance deteriorates as we construct more PDGs. One of the possible future work is to optimize the PDG construction.

In our experiments we presented the cases our tool handles and for which cases it falls short. Our tool does not analyze the content access for pointers which would reveal that they are `NULL` terminated C strings for some cases in our experiments. Combined with adding more functions in our pre populated lists, adding access analysis on PDG would improve our results. Additionally, we only analyze `malloc` function calls and static arrays for `[count]` attribute. Covering other allocation techniques in our static analysis increases the precision of our heuristics. Finally, our results show that full PDG analysis for each and every local variable in the application would further improve our inference quality.

5.1 Limitations

SGX incurs several limitations in C/C++ application such as the unsupported library functions we mentioned earlier. While we handle these functions with our library of wrappers, there are also some structures and types that are unsupported such as `FILE` pointer. Our tool does not automatically convert these into `SGX_FILE` secure type introduced in SGX. More detail about unsupported libraries keywords and functions can be found in [3].

Bibliography

[1] Costan, V. and S. Devadas (2016) “Intel SGX Explained.” *IACR Cryptology ePrint Archive*, **2016**(086), pp. 1–118.

[2] Huang, Y. (2019) “Automatic IDL Generation for Privilege Separation,” .

[3] Intel, R. (2016), “Software Guard Extensions SDK Developer Reference for Linux* OS,” .

[4] Ferrante, J., K. J. Ottenstein, and J. D. Warren (1987) “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **9**(3), pp. 319–349.

[5] Liu, S., G. Tan, and T. Jaeger (2017) “PtrSplit: Supporting general pointers in automatic program partitioning,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2359–2371.